A COMPUTER HARDWARE DESIGN LANGUAGE FOR MULTIPROCESSOR SYSTEMS

BY

TREVOR NIGEL MUDGE

B.Sc., University of Reading, 1969
M.S., University of Illinois, 1973

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1977

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

August, 1977

WE HEREBY RECOMMEND THAT THE THESIS BY

TREVOR NIGEL MUDGE

ENTITLED A COMPUTER HARDWARE DESIGN LANGUAGE FOR MULTIPROCESSOR SYSTEMS

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

Director of Thesis Research

Head of Department

Committee on Final Examination†

Chairman

William J. Kubitz

H George Friedman Jr.

Wilfred J. Hansen

Jacob A. Abraham

† Required for doctor's degree but not for master's

0-517

ACKNOWLEDGMENT

I would like to thank my advisor Professor Gernot Metze for his support and guidance, and James Smith, Ravi Nair, and B. Kumar for many useful discussions. Further, I would like to thank Janet Van Weringh and Janet Van Valkenburg for typing the rough draft, Mrs. H. Corray for typing the final draft and Robert MacFarlane and Alan Wier for producing the drawings. Finally I would like to thank Jean Dussault, Scott Woodard, Alan Gant, Joel Emer, William Kaminsky, Daniel Hammerstrom, William Davidson, Satish Thatte, and Professors Edward Davidson, Jacob Abraham and Richard Flower for making my stay in the Digital Systems Group an enjoyable one.

"An ounce of prevention is
worth a pound of cure."
-Proverb

TABLE OF CONTENTS

1. INTRODUCTION

In an attempt to formalize the design process for large digital systems, many researchers have suggested the use of computer hardware design languages (CHDLs)*. However, using a CHDL does not necessarily facilitate the design process. An ill conceived language can encumber the design process and fail to guide it away from design errors. Such a CHDL then becomes useful only as a documentation aid. It is our belief that most CHDLs fall into this category, and it is interesting to note that one of the most popular CHDLs, called ISP [Bel 71], started out as such.

The two purposes of this thesis are:

1. To develop a CHDL with sufficient scope to describe multiprocessing systems.
2. To specify the CHDL so that syntactically correct programs describe systems which have deadlock-free control structures (CSs).

The control problem associated with multiprocessing systems is, in general, quite complex, and the opportunities for creating a CS which can hang-up are great. Specifying the CHDL so that this pitfall can be avoided by staying within the bounds of the syntax, gives the user a true design tool which is more than just an aid for documenting the principles of operation of a system.

The two aims stated above are to some extent opposing. The first requires that the CHDL have many constructs, and the second that it have few (if the view is taken that restricting the language also restricts its ability to describe undesired objects). However, any compromise reached

---

*Two comprehensive guides to literature on this topic are [Fig 73] and [Bar 75]. A recent collection of papers can also be found in [Pro 75].

is bound to be unsatisfactory from some viewpoint, and in our case a very similar situation exists to the one highlighted by Knuth in [Knu 74], concerning goto-less structured programming. He points out that although goto-less structured programming retains completeness while enhancing the potential for error-free programming, some algorithms can only be realized in a clumsy way. By analogy, although our CHDL is in some sense complete and aids error-free design, some control algorithms can only be realized in a clumsy way.

1.1 The System Model Presumed by the CHDL

In multiprocessing where there are often several independent processes active simultaneously that must be coordinated and synchronized without being unnecessarily bound together, the most natural model for a CS is an asynchronous one. This is the one we have used.

The CHDL is capable of being used to design digital systems which conform to the following model: the system partitions into a hierarchically organized CS and a data structure (DS). Actions in the DS are assumed to be representable as register-transfers. The coordination of these actions is accomplished by the CS. The register transfers themselves are initiated by request (R) signals, which issue from the CS and travel over bidirectional signal paths called links to the DS. Upon their completion acknowledge (A) signals are transmitted back along the links to the CS. To enable the CS to test bit values in the DS, a second type of link, called a conditional link, is needed. These links carry three signals; a test (T) signal that goes from the CS to the bit to be tested, and two result signals (I_1 and I_0), one of which is transmitted back along the link to the CS, depending on whether the bit was 1 or 0. The system model is shown in Figure 1.1.

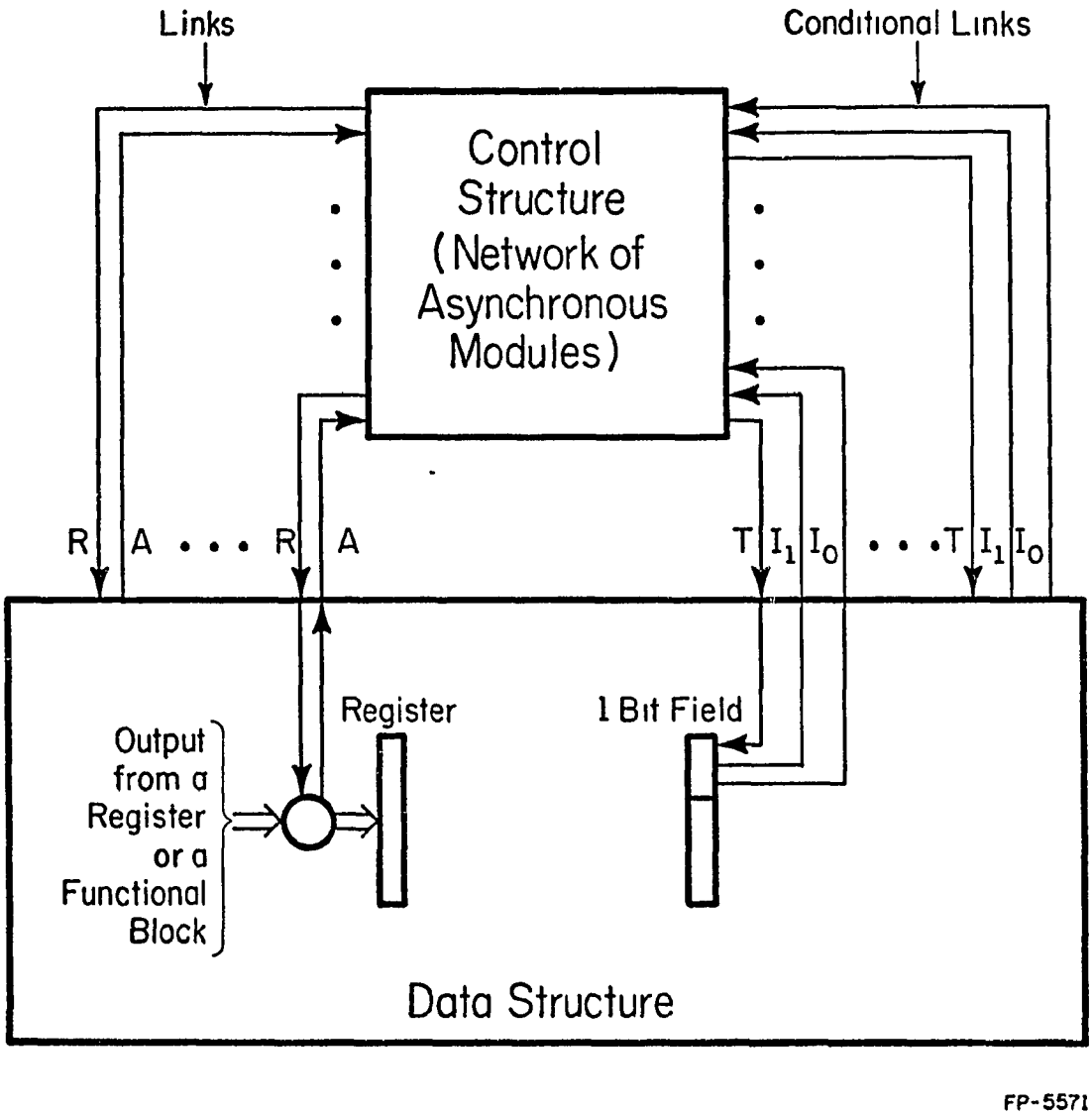


Figure 1.1. The System Model.

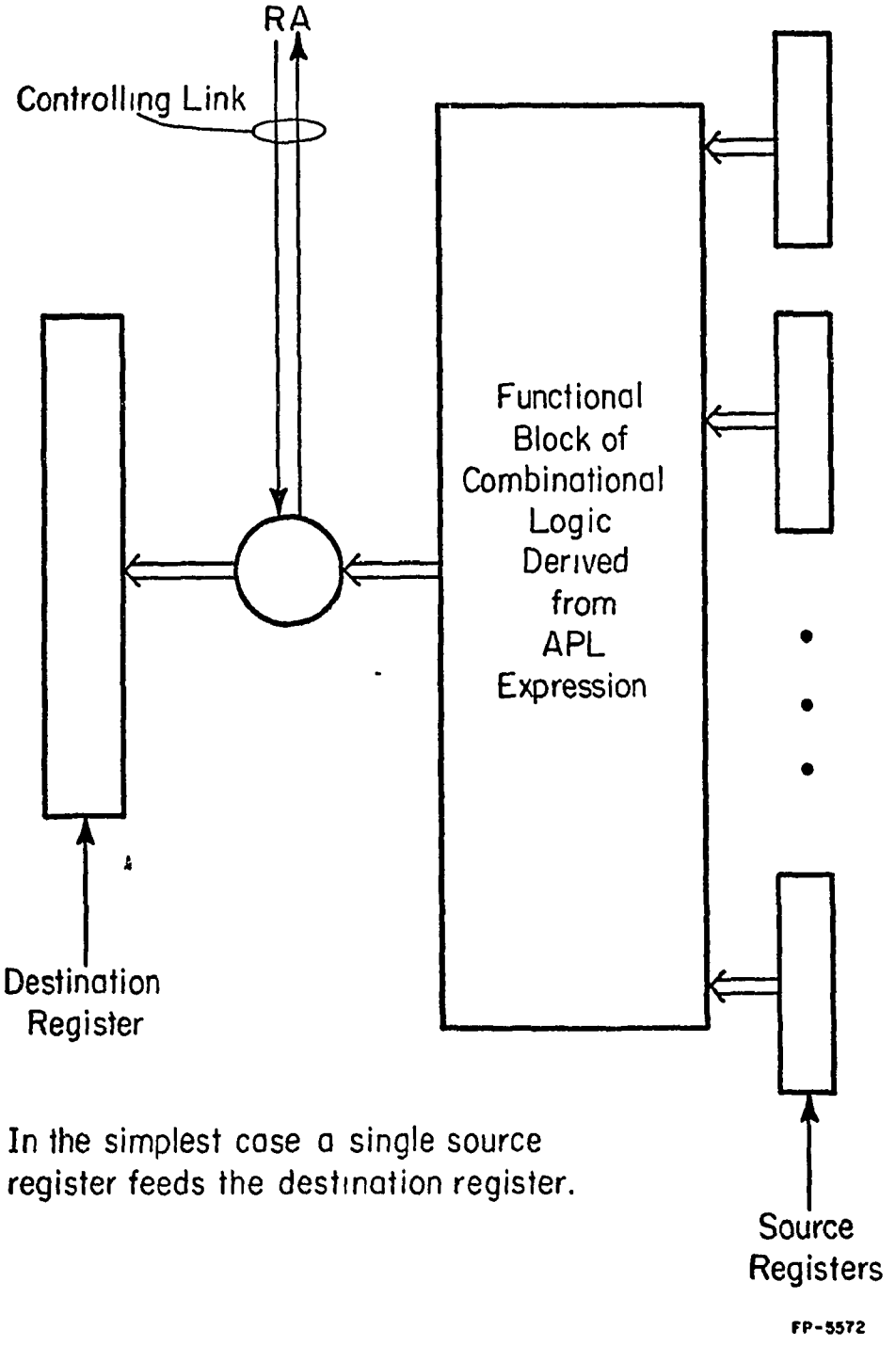


Figure 1.2. Register-transfer Logic.

The CHDL translates to a collection of asynchronous CS modules interconnected by links to form a network which constitutes the CS of the target system. There are ten different types of CS modules, any number of which can be used to form the network.

The register-transfers controlled by these networks are also described by the CHDL. They have the form

D ← S

where D is the name of a destination register, and S is an APL expression whose arguments are taken to be registers in the DS. From the standpoint of the CS these expressions can be regarded as functional blocks of combinational logic. Figure 1.2 illustrates a register-transfer in more detail. The actual structure and design of the functional block is not specified by the CHDL. That is assumed to be taken care of off-line, possibly by another program which forms part of a CAD set-up. Such programs are discussed by Friedman in [Fri 67] and [Fri 69].

To describe the operation of systems conforming to this model it is convenient to use the undefined term "process". This is meant to be some activity in the target system that is initiated with a request signal and terminates with an acknowledge signal. The operation of any system specified by the CHDL can then be regarded as a process which decomposes into other less complex processes. These in turn decompose until finally the operation of the system can be viewed as a collection of atomic processes - the register-transfers that are coordinated by the CS. This hierarchical structuring of processes corresponds to the hierarchical organization of the CS. Communications over links between hierarchical levels in the CS correspond to the initiation and termination of processes. We shall see later that the different levels of control are a natural consequence of the block structured nature of the CHDL.

1.2 The Plan of the Thesis

This thesis is arranged as follows.

Chapter 2 introduces the ten CS modules and defines their behaviors using Petri net graphs. It also gives rules for interconnecting these graphs so that the behavior of networks of CS modules can be deduced.

Chapter 3 presents the syntax of the CHDL as a set of production rules, together with some terminology to enable later discussion about objects in the syntax.

Chapter 4 gives an interpretation of the CHDL in terms of process behavior, and a procedure for translating programs in the CHDL into networks of CS modules. These two things are related using the Petri net graphs of Chapter 2.

Chapter 5 illustrates the use of the CHDL by presenting the design of a small system. The system is a processor which executes register-to-register instructions. These operate on a DS of four registers and two multi-purpose function units. The CS is implemented as a forwarding algorithm to achieve instruction execution look-ahead. Such an example has many of the control requirements of a typical multiprocessor system. Thus it illustrates well the capabilities of the CHDL.

Chapter 6 discusses the scope of the CHDL. Due to the acknowledged scope of APL to characterize the functional aspects of the DS, the scope of the CHDL is examined from a CS viewpoint. An indication of its completeness is made, and it is concluded that the first purpose of this thesis has been met.

Chapter 7 introduces some additional syntactic requirements. Then it is proved, using a method for characterizing the behavior of networks of CHDL blocks, that syntactically correct CHDL programs (i.e. ones that

satisfy the syntax of Chapter 3 plus the additional syntactic requirements) describe systems which have deadlock-free CSs. Computational complexity arguments show that checking the syntax (excluding the APL expressions of the register-transfers) is very simple. Thus, freedom from deadlock can be achieved without complicating the syntax of a CHDL or limiting its scope (this last point from Chapter 6). It is concluded that the second purpose of this thesis has been met, without resorting to a complex syntax.

Chapter 8 discusses two approaches to the implementation of the CHDL programs in hardware. The first, based on the asynchronous model of Chapter 1, discusses constructing the CS modules from logic gates and then constructing the functional blocks of the DS with additional logic to generate acknowledge signals. The second discusses a very natural synchronous realization, which employs a finite state machine for the CS (realizable as a PLA and a set of flip-flops) and a bus structured DS. This approach is shown to overcome the drawbacks associated with requiring parts of the DS to generate acknowledge signals, while retaining many of the advantages of an asynchronous CS.

Chapter 9 mentions other applications of some of the ideas mentioned in this thesis and compares our approach to CHDLs with others.

Finally in Chapter 10 some concluding remarks and suggestions for further research are made.

2. BEHAVIORAL DESCRIPTIONS OF THE CS MODULES

In this chapter the behaviors of the CS modules are defined. (The actual implementation of these behaviors is not discussed until Chapter 5.) The behavior for each module is defined by means of a Petri net (PN) graph [Pet 66] [Hol 68], and rules are given for interconnecting these behavior graphs so that the behavior of networks of interconnected CS modules can be deduced.

2.1 The Petri Net Graph

The following definition of a PN is similar to that found in [Hei 76].

A PN is a four-tuple $<P, T, A, M_o>$ where

P is a non-empty set of distinctly labelled places

$$\{p_1,\ldots,p_n\}$$

T is a non-empty set of distinctly labelled transitions

$$\{t_1,\ldots,t_m\}$$

A is a relation, $A \subseteq (P \times T) \cup (T \times P)$

M_o is the initial marking.

A marking, M, for a PN is a function $M: P \rightarrow Z$, where Z is the set of non-negative integers. M(p) is referred to as the token load of the place p or as the number of tokens on p.

PNs are conveniently represented as directed graphs. Places and transitions are the nodes of the graph and the directed arcs show the relation A. The graph is bipartite since each arc connects a place (or transition) to a transition (or place). Tokens are represented as dots in the place nodes. If p_i is a place and t_j is a transition and if $<p_i,t_j>$ belongs to A, then p_i is an input place of t_j and t_j is an output transition of p_i. Similarily, if $<t_j,p_i>$ belongs to A, t_j is an input transition of p_i and p_i is an output place of t_j.

Figure 2.1 shows an example of a PN. For this example, the relation A is:

$$A = \{\langle p_1,t_1\rangle,\langle p_4,t_1\rangle,\langle p_2,t_2\rangle,\langle p_5,t_2\rangle,\langle p_3,t_3\rangle$$
$$\langle t_1,p_2\rangle,\langle t_2,p_3\rangle,\langle t_3,p_4\rangle,\langle t_3,p_1\rangle,\langle t_3,p_5\rangle\}$$

The marking shown has value 1 for places p_1, p_4 and p_5 and value 0 for places p_2 and p_3.

So far we have defined the static properties of PNs. Next we define the dynamic properties of PNs. It is the dynamic quality of PNs that make them ideal models for asynchronous processes.

A transition in a PN is enabled if each of its input places contains a token. An enabled transition can fire, which transforms the marking of the net by removing one token from each input place of the transition and adding one token to each output place of the transition. Clearly, a sequence of transition firings, a firing sequence, causes a sequence of marking transformations.

The following procedure which characterizes the dynamic quality of a PN is called simulation.

1. Compute the set of enabled transitions (U).
2. Choose one transition $t_i \in U$.
3. Fire t_i.
4. Go to 1.

Consider the example of Figure 2.1. If the markings are represented as vectors of length 5, then the marking shown is (1,0,0,1,1) where the order from left to right is p_1, p_2, p_3, p_4, p_5. Simulating this simple example generates a single cyclic firing sequence:

$$(1,0,0,1,1) \xrightarrow{t_1} (0,1,0,0,1) \xrightarrow{t_2} (0,0,1,00) \xrightarrow{t_3} (1,0,0,1,1)$$

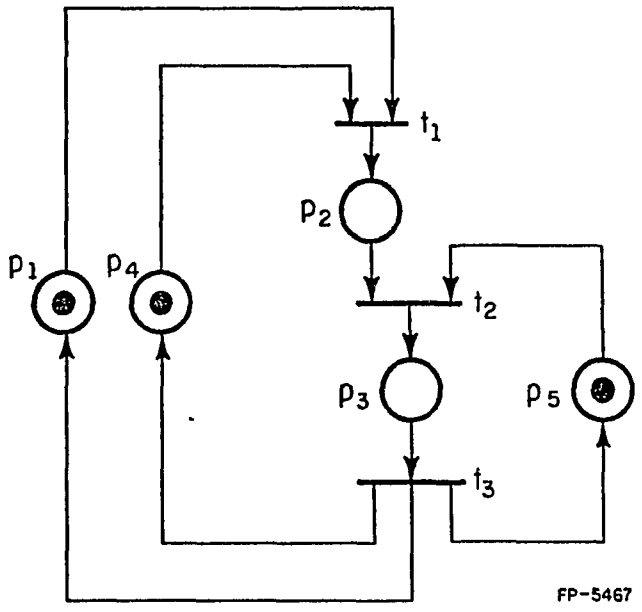


Figure 2.1. An Example PN.

The notation $M \xrightarrow{t} M'$ is meant to indicate that firing transition t transforms the marking M into M'. This notation can be extended to sequences of transitions, leading to the following definitions:

A marking M_j is <u>reachable</u> from M_i if $\exists$ a sequence $\sigma \in T^* \ni M_i \xrightarrow{\sigma} M_j$

The <u>forward marking class</u> $\vec{M}$ of a marking M is the set of markings reachable from M.

$$\vec{M} = \{M' \mid \exists \sigma \in T^* \text{ and } M \xrightarrow{\sigma} M'\}$$

A transition (place) is <u>dead</u> for the marking M if $\forall M' \in \vec{M}$, the transition (place) is disabled (does not contain a token).

A PN = $<P, T, A, M_o>$ is <u>safe</u> if $M(p) \leq 1 \quad \forall p \in P$ and $\forall M \in \vec{M}_o$

A PN is <u>live</u> if $\forall M \in \vec{M}_o$ no transition (place) is dead

These last two definitions will be used later (in section 7) to define a deadlock-free CS.

Figure 2.2 shows how we will use PNs to model processes in digital systems. Processes are associated with places, and their occurrence with tokens in those places. In the example of Figure 2.2 the onset of process P is indicated by the firing of transition R. This causes a token to be deposited in p. The presence of a token in p indicates the occurrence of process P. The termination of P is indicated by the firing of A, and the resulting removal of the token from p.

The labels R and A for the transitions which demark the process P were intentionally chosen to correspond to the request and acknowledge signals used on the links postulated in the system model of section 1. Now it can be seen how PNs can be used to model the behavior of digital processes controlled by links: the transmission of a request signal from

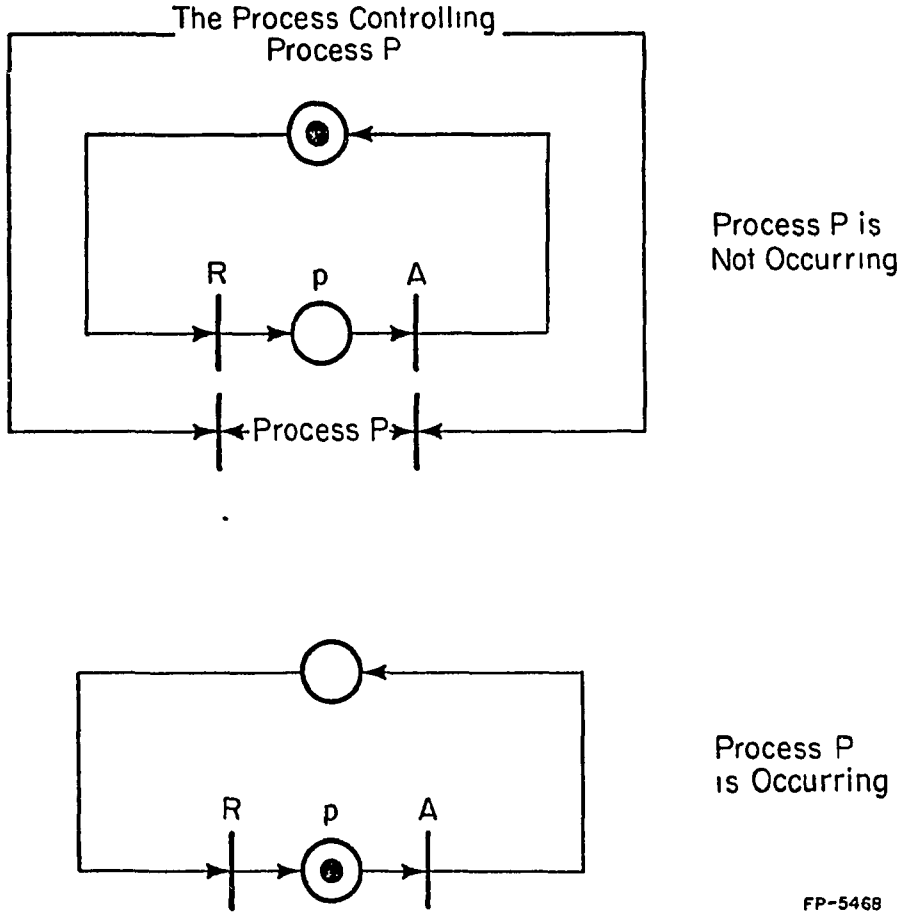


Figure 2.2. A PN Model of a Process.

the controlling process down the controlling link corresponds to the firing of the transition R, and when the process (P) controlled by the link is completed, the transmission of the acknowledge signal up the link corresponds to the firing of transition A. (The link is viewed as an output link by the controlling process, and an input link by process P.)

Since a single link may control a complete subsystem, the occurrence of a process, such as P in our example, may be interpreted as the initiation, running and termination of a subsystem, which itself may be composed of a collection of other processes. If the PN which defined the behavior of this collection of processes, or subsystem, were substituted for P, a new PN would result representing a more detailed account of the system behavior.

We are now in a position to define the behaviors for the ten CS modules.

2.2 The Source Module

The source (So) module is shown in Figure 2.3. On the left is a diagrammatic representation. It has one output link, which is shown as an input acknowledge (A) signal line and an output request (R) signal line. On the right is a more concise diagrammatic representation of the module. This time the link is represented by a single directed arc, directed in the direction in which the request signal travels. (This last convention will be used throughout the remainder of this discussion.) In the center is the PN graph of the So module with its initial marking. By simulating this PN the behavior of the module can be deduced. The occurrence of a request signal on the request signal line is indicated by firing transition R, and the occurrence of an acknowledge signal line is indicated by firing transition A. (In later sections subscripts are used to denote

the link to which the signal belongs.) From the PN it can be seen that the module transmits a request signal initially, and then retransmits a request signal whenever an acknowledge signal is received. It thus acts as a source of requests.

2.3 The Sink Module

The sink (Si) module is shown in Figure 2.4. In operation it complements the So module. Whenever it receives a request signal it transmits an acknowledge signal. It thus acts as a sink for requests. Notice its PN is identical to that for the So module. However, in the So module the request is an output signal and the acknowledge an input signal. In the Si module the converse is true.

2.4 The Wye Module

The wye (W) module is shown in Figure 2.5. By simulating the PN it can be seen that when a request is received on link 1 (R_1), requests are transmitted on links 2 (R_2) and 3 (R_3) both. When an acknowledge signal is received on both links 2 (A_2) and 3 (A_3) (in any order) an acknowledge is transmitted on link 1 (A_1). Thus a W module may be used by a process to simultaneously initiate two other processes. Only when both of these processes are completed (i.e. when the module has received acknowledge signals on links 2 and 3) is the controlling process notified by the transmittal of an acknowledge along link 1.

2.5 The Sequence Module

The sequence (S) module is shown in Figure 2.6. By simulating the PN it can be seen that when a request is received on link 1 a request is transmitted on link 2. When an acknowledge is received on link 2 a request is transmitted on link 3. Finally an acknowledge on link 3 causes

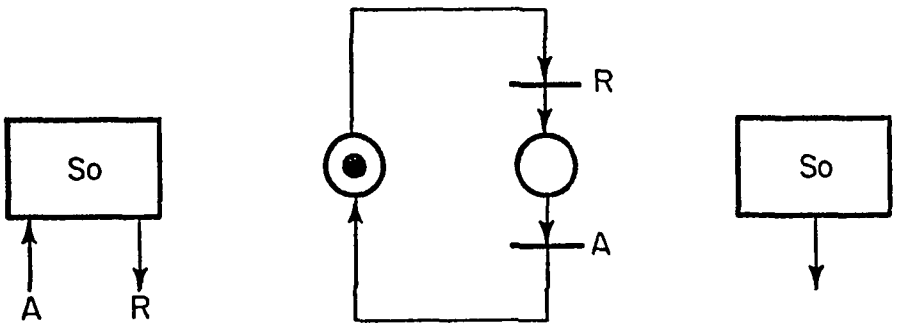


Figure 2.3. The Source Module.

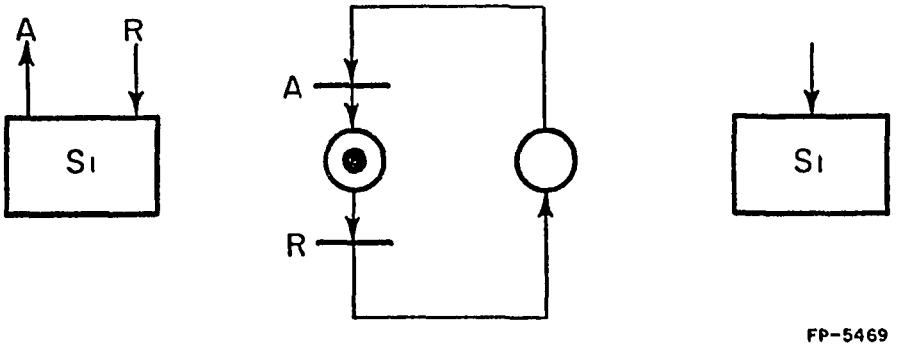


Figure 2.4. The Sink Module.

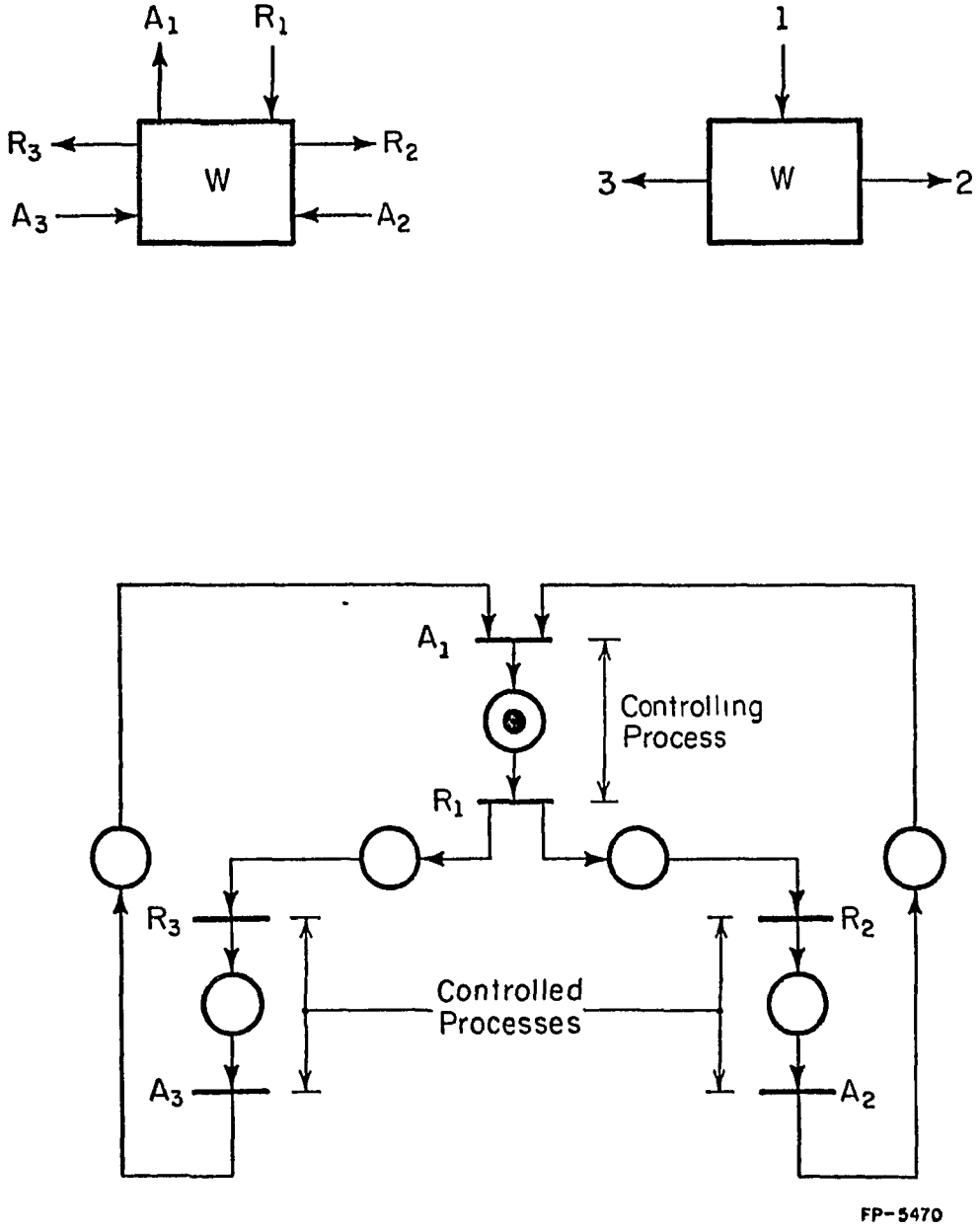


Figure 2.5. The Wye Module.

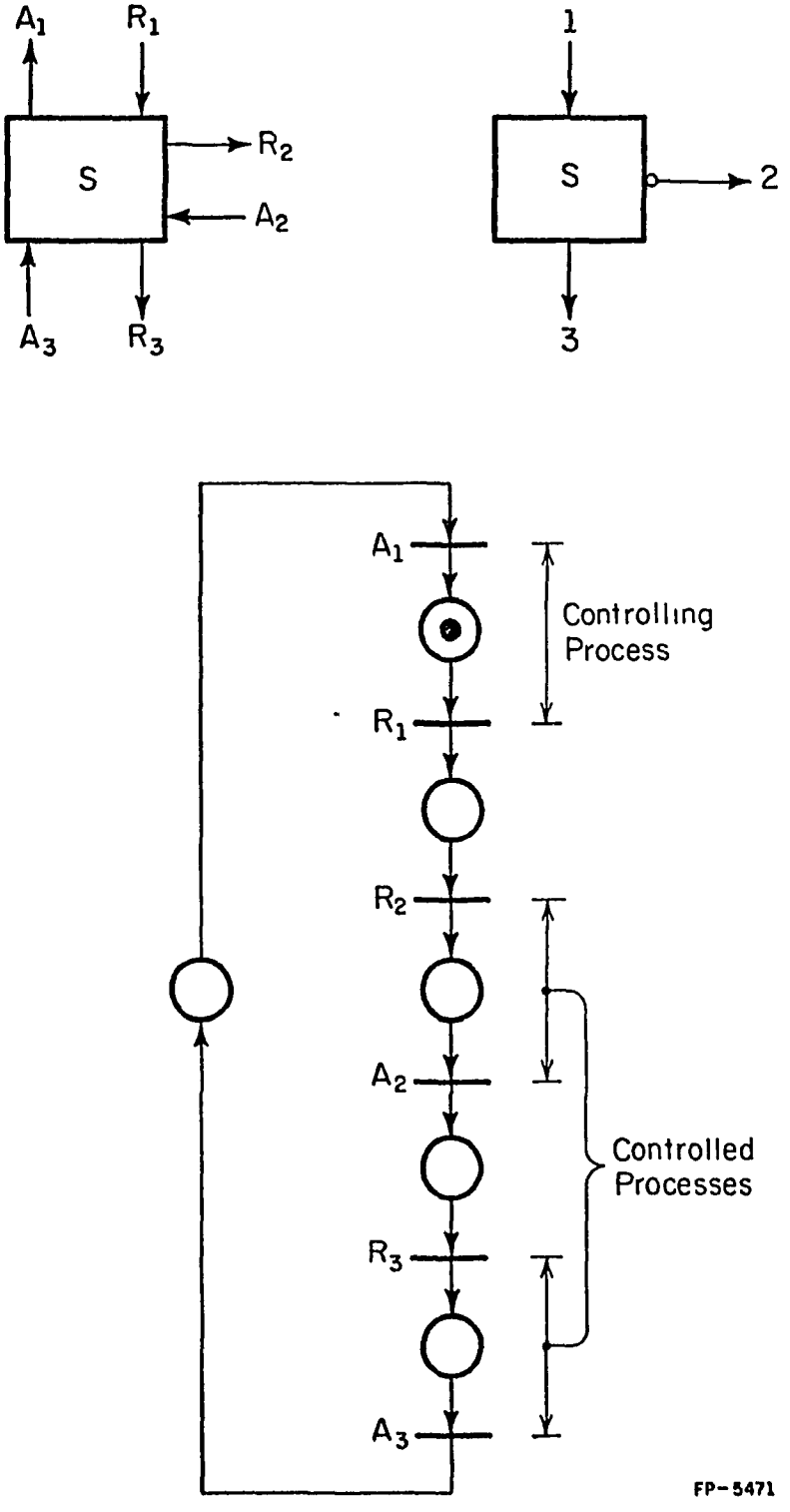


Figure 2.6. The Sequence Module.

an acknowledge to be transmitted on link 1. Thus the S module may be used by a process to initiate two processes one after the other. The controlling process requests on link 1 whereupon the process controlled by link 2 is performed. On its completion the process controlled by link 3 is performed, and an acknowledge is returned to the controlling process.

The temporal sequencing between the processes controlled by links 2 and 3 is indicated in the diagrammatic representation of the module at the top right of Figure 2.6. Link 2 is shown with a circle at its base, indicating that the process that it controls precedes in time that controlled by link 3. These two links are called the primary and secondary output links of the S module.

2.6 The Trigger Module

The trigger (T) module is shown in Figure 2.7. By simulating the PN it can be seen that its behavior is similar to that of the S module, except that control is returned to the controlling process as soon as the process controlled by link 2 is completed. Hence the controlling process and the process controlled by link 3 can overlap in time (they can both have tokens in their respective places). However, the controlling process can never get more than one occurrence ahead of process 3 (we shall adopt the convention of labelling processes the same as the links associated with them, unless otherwise indicated), as process 2 cannot be reinitiated until process 3 is completed.

Thus the T module implements the basic control mechanism for an assembly-line station platform (called a trigger [And 67]), in a chain of processes that process data in a pipeline, or assembly-line fashion.

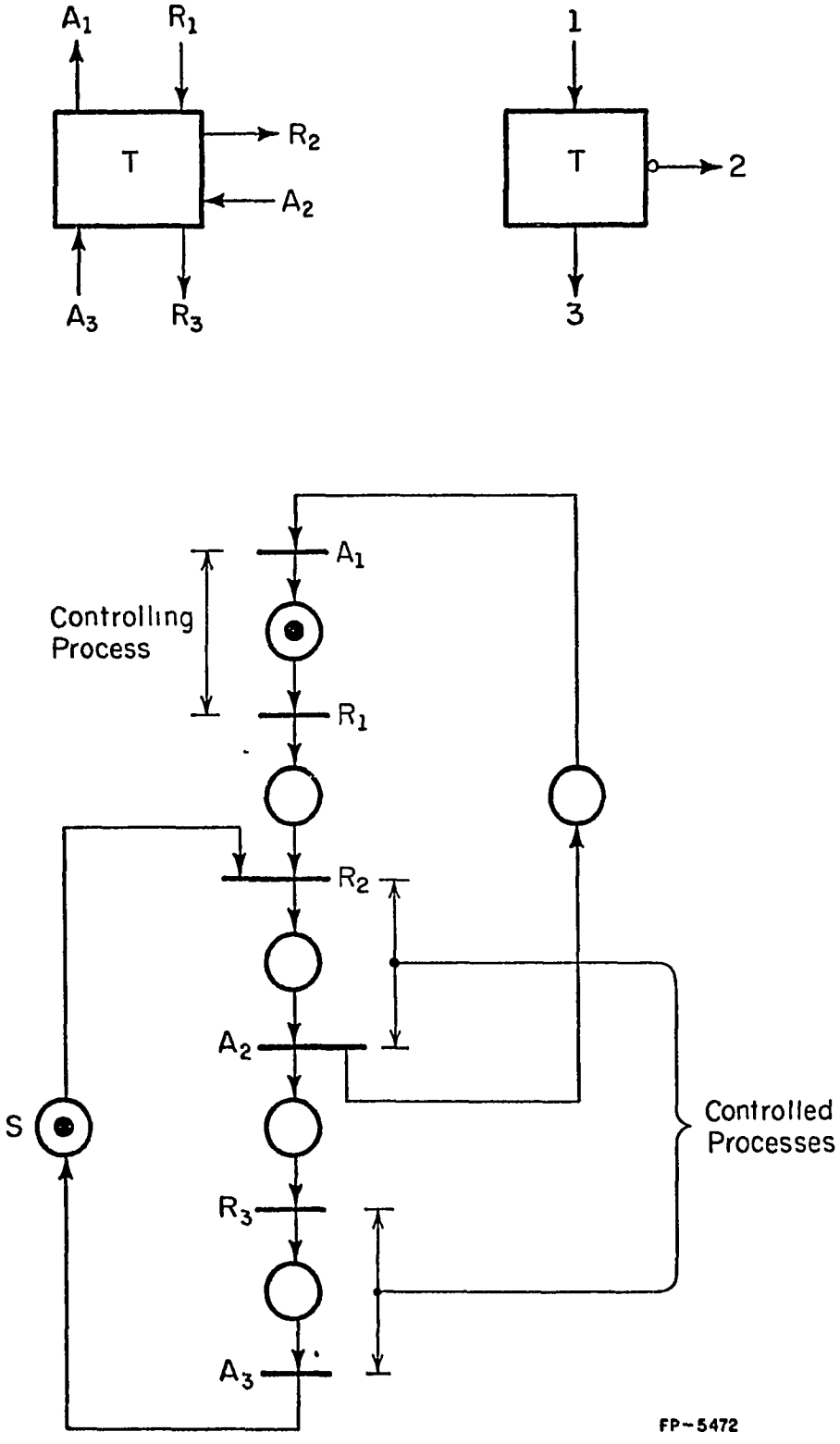


Figure 2.7. The Trigger Module.

2.7 The Junction Module

The junction (J) module is shown in Figure 2.8. Its operation can be viewed as the dual of the W module. It may be used by two controlling processes to initiate a third process. The controlling processes request over links 1 and 2. The third process is controlled by link 3, and is not initiated until both the controlling processes have requested it. The module thus performs an act of synchronization between two concurrent processes, before initiating a third. When the controlled process is completed it broadcasts an acknowledge to both the controlling processes.

2.8 The Shared Resource Module

The shared resource (SR) module is shown in Figure 2.9. It can be thought of as a module for allowing two processes to share some other process (their common resource).

If a request is received on link 1 then process 3 is initiated. When this is completed an acknowledge is received on link 3 and an acknowledge is transmitted along link 1. Similarly if the request is received on link 2. Thus either of the controlling processes can gain control of process 3. If requests on link 1 and 2 overlap (i.e. requests occur on links 1 and 2 without an intervening acknowledge on link 3) they are still handled in the order in which they arrive. If they occur simultaneously they are handled in arbitrary order.

2.9 The Mutual Exclusion Module

The mutual exclusion (ME) module is shown in Figure 2.10. Controlling process 1 can gain control of process 3, and controlling process 2 can gain control of process 4. The module imposes mutual exclusion on these two otherwise unrelated transfers of control. In other words, if process 1 has

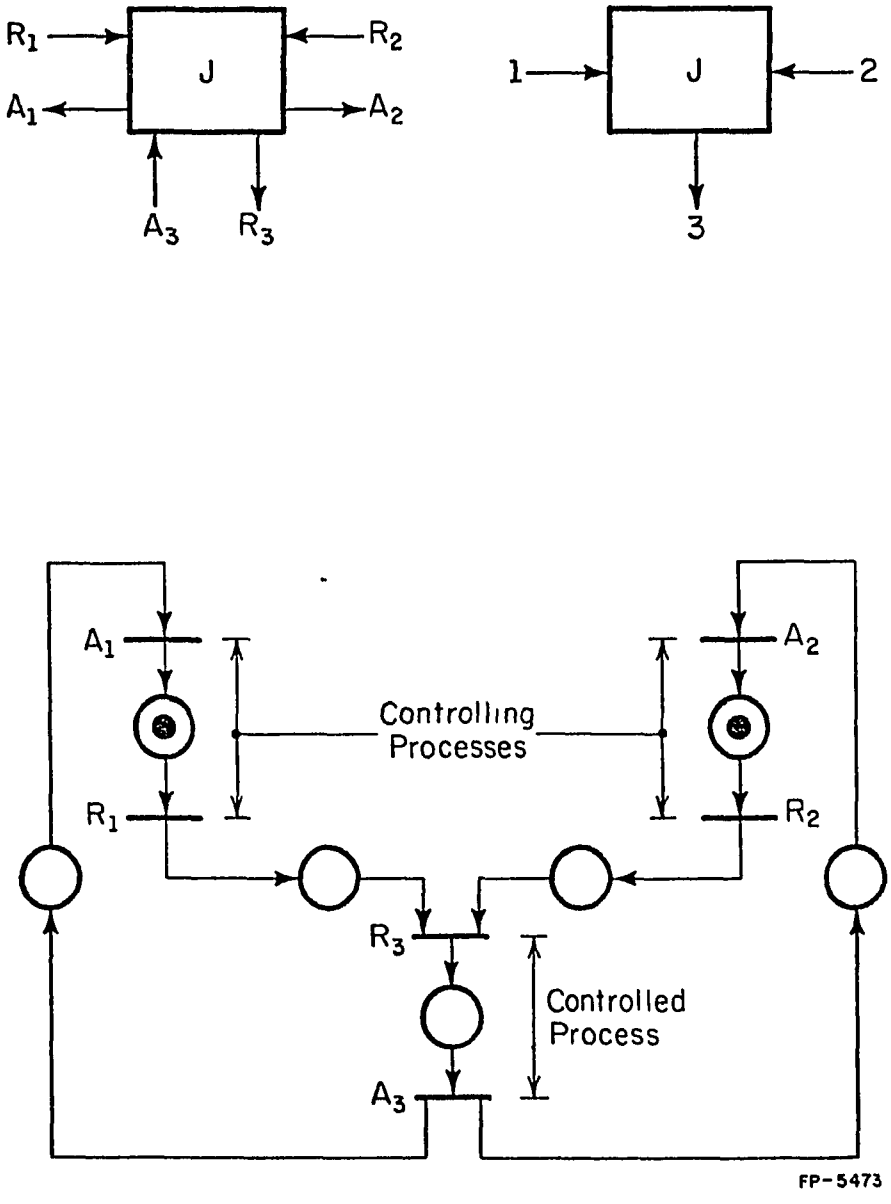
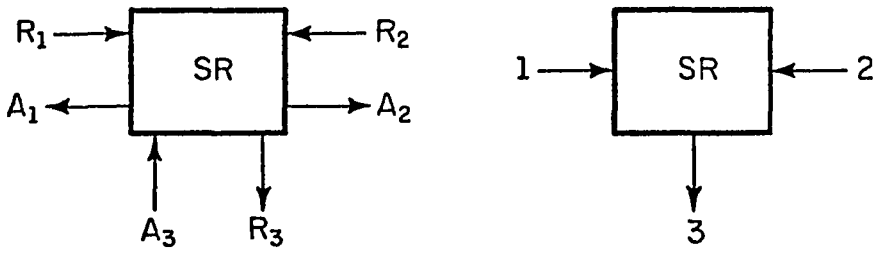


Figure 2.8. The Junction Module.



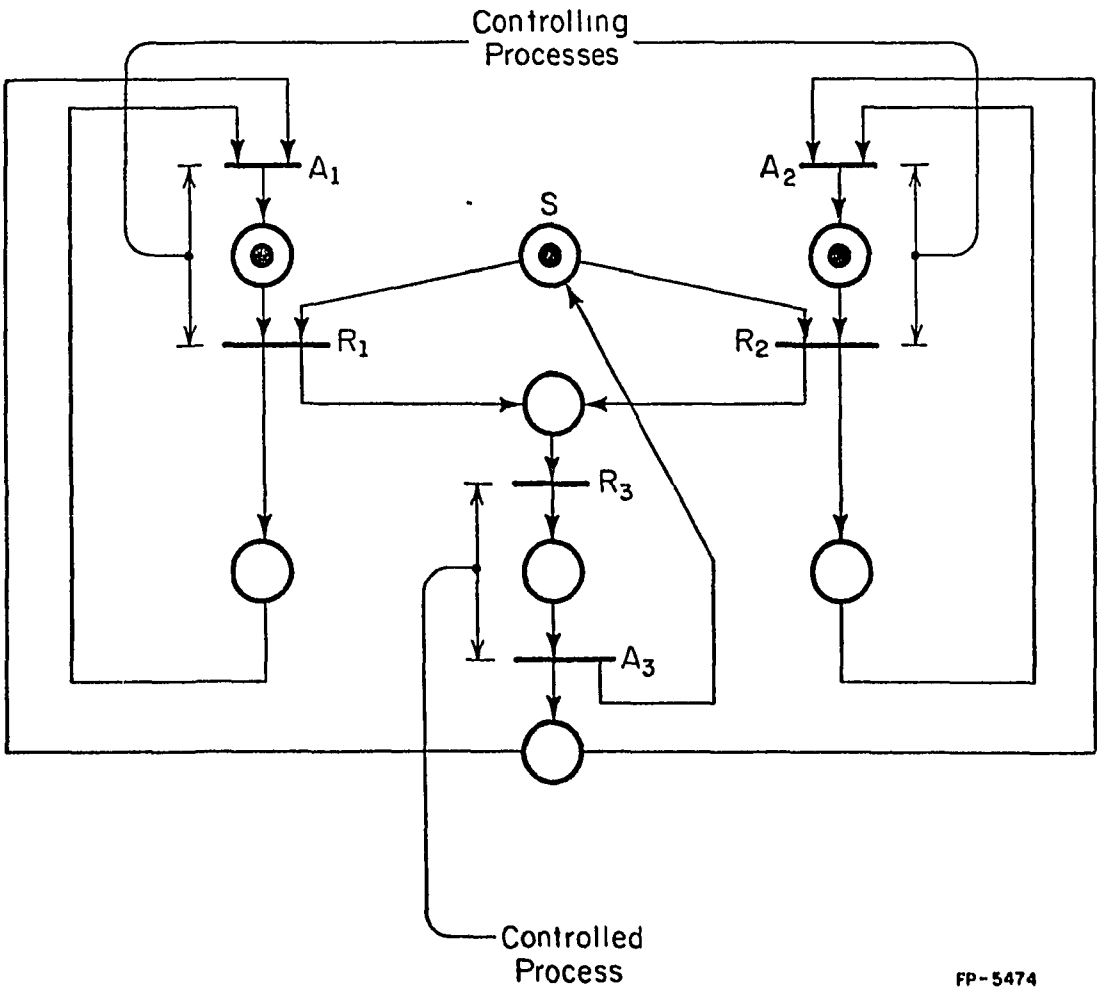
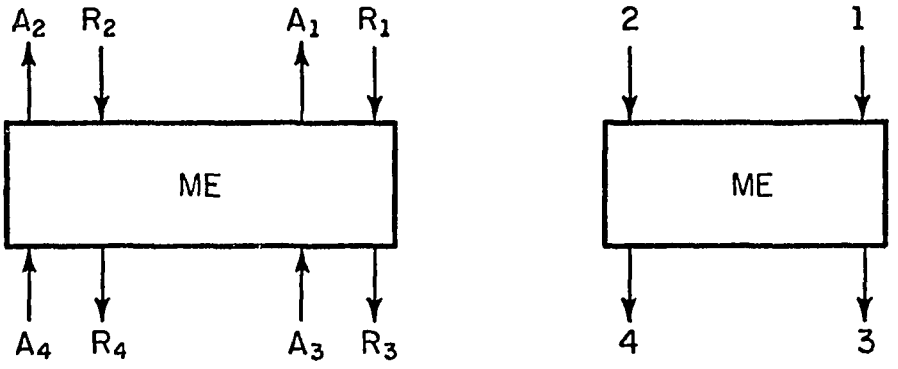


Figure 2.9. The Shared Resource Module.



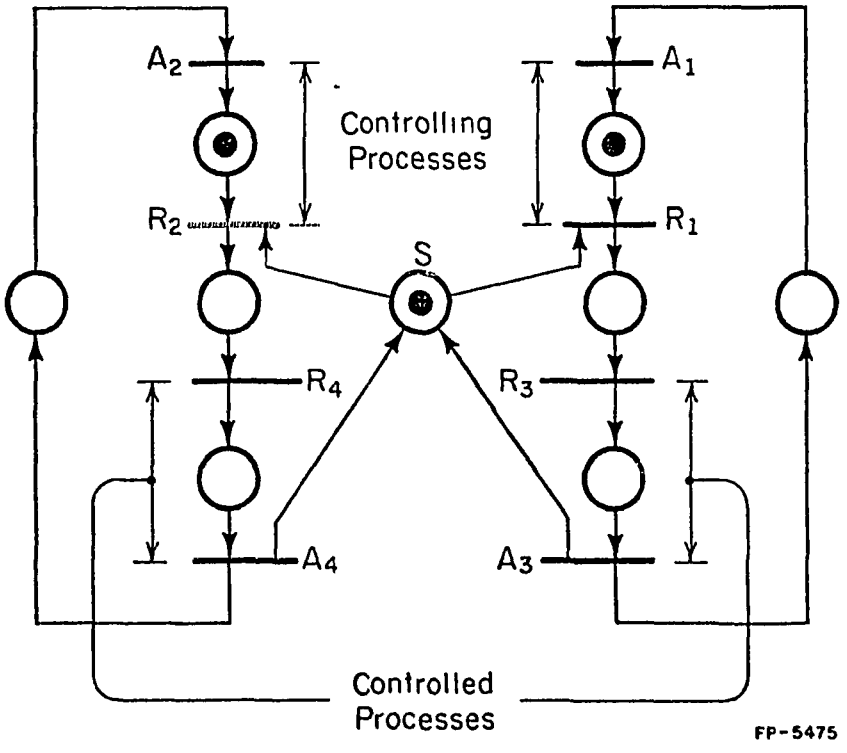


Figure 2.10. The Mutual Exclusion Module.

control of process 3, 2 cannot gain control of 4 until 1 releases 3. If both processes 1 and 2 simultaneously seek control of processes 3 and 4 respectively, then one pair is preferred and it is chosen arbitrarily.

The ME module allows two processes (1 and 2) to share common parts of the DS (controlled by processes 3 and 4 respectively) while maintaining the determinism of those processes.

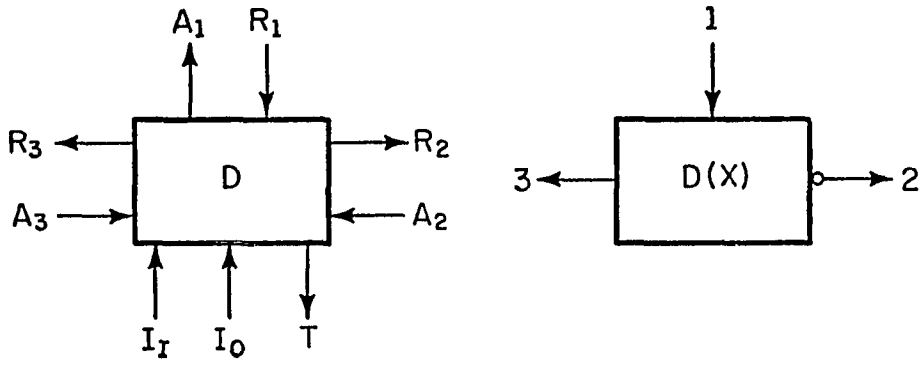
Both the ME module and the SR module exhibit mutual exclusion between two processes. This behavior is achieved by the place S (see Figure 2.9 and Figure 2.10) which is analogous to a binary semaphore initially set to 1.

2.10 The Decode Module

The decode (D) module is shown in Figure 2.11. A controlling process requests on link 1. This request is transmitted on link 3 or 2 depending on whether the external boolean variable x is 1 or 0. The acknowledge is returned in the usual manner. Thus the D module may be used as a branch point in a CS. The branch is controlled by the bit x.

In the diagrammatic representation of the D module at the top left of Figure 2.11 the link used to test x is shown as a conditional link. A signal is transmitted on line T to test the bit and is returned on either I_1 or I_0 depending on whether x is 1 or 0. The testing occurs every time the module receives a request on link 1. The more concise diagrammatic representation of the module in the top right of Figure 2.11 distinguishes the link through which control flows if x is 0 by the circle at its base. The module is labelled D(x) to identify its function (decode) and the name of its argument (x in this case).

The signals of the conditional link are not explicitly modelled as transitions in the PN graph. Instead the test and its result are modelled by a free-choice node, place f (see [Pat 72] for further explanation of the



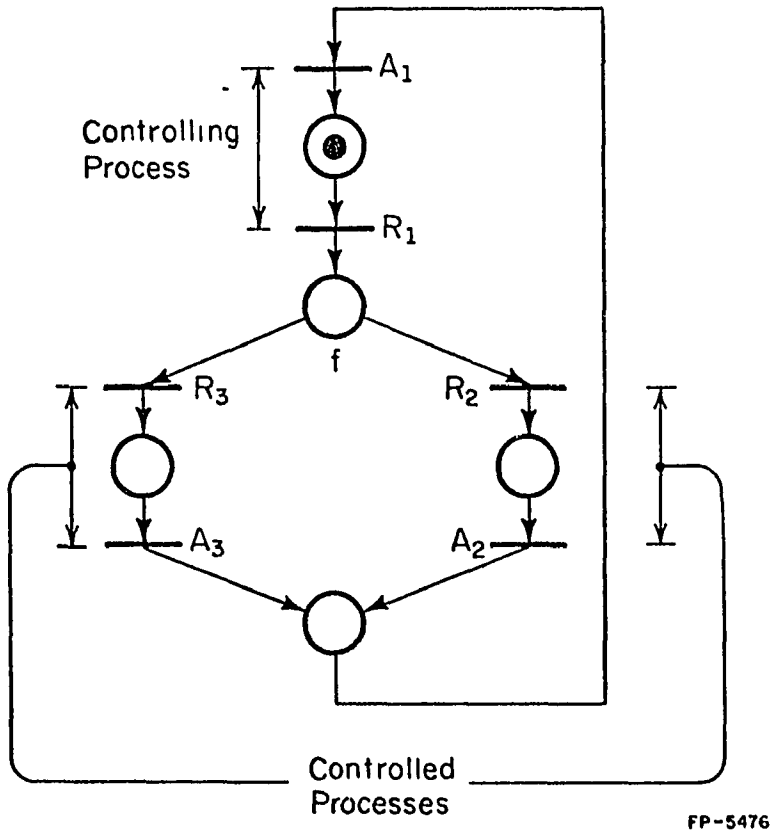


Figure 2.11. The Decode Module.

term free-choice). A token in f can fire either R_2 or R_3 but not both. This allows for both possible mutually exclusive outcomes of the test.

2.11 The Iterate Module

The iterate (I) module is shown in Figure 2.12. A controlling process requests on link I. If the value of the external boolean variable x is 0 an acknowledge is transmitted back along link 1. If x is 1 the process controlled by link 2 is initiated by transmitting a request on link 2. When process 2 is completed an acknowledge is received on link 2, and if x is still 1 a request is retransmitted on link 2 reinitiating process 2. This reinitiation continues as long as x is 1. If an acknowledge is received on link 2 when x is 0, process 2 is no longer reinitiated. Instead an acknowledge is transmitted on link 1 back to the controlling process. This module may be used in a CS when a process is required to be reinitiated as long as some external bit is 1.

The link used to test x is a conditional one, similar to the one used by the D module. It is also modelled by a free-choice node (place f). Similar to the D module, the more concise diagrammatic representation of the module shown at the top right of Figure 2.12 is labelled I(x) to identify its function (iterate) and the name of its argument (x in this case).

2.12 The Behavior of Networks of CS Modules

We are now ready to present an algorithm, which allows us to construct the PN graph representing the behavior of networks of CS modules, from the PNs of the individual nodules given in the last ten subsections.

Two cases must be taken into account by the algorithm. In the first, an output link of one network of modules is connected to an input link of

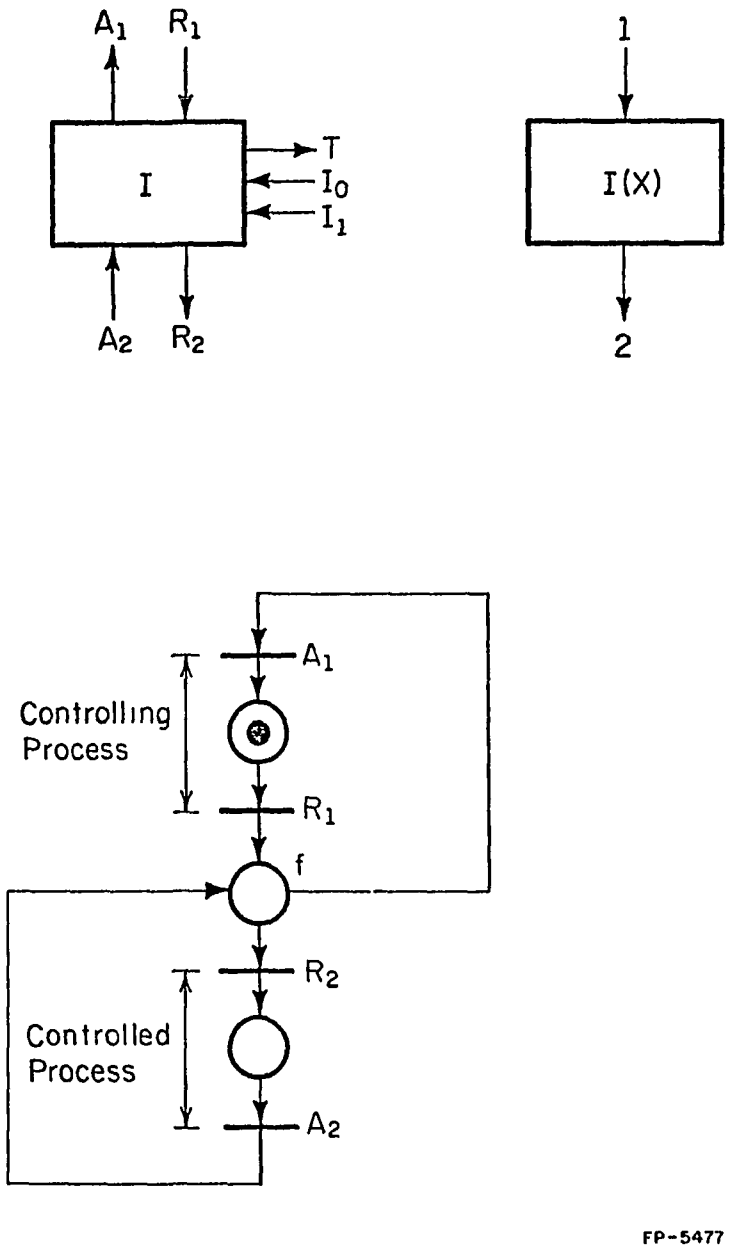


Figure 2.12. The Iterate Module.

another network of modules, to form a larger single network. In the second, an output link of a network is connected to an input link of that same network, to produce a slightly different network.

In both cases, the construction algorithm can be described informally as follows (see Figure 2.13; PN_1 may be the same as PN_2):

1. Discard p and q together with their input and output arcs.
2. Combine R_1 and R_2 into a new transition, λ_R, such that the input arcs to λ_R are those that were inputs to R_1 and the output arcs are those that were outputs of R_2.
3. Combine A_1 and A_2 into a new transition, λ_A, such that the input arcs to λ_A are those that were inputs to A_2 and the output arcs are those that were outputs of A_1.

More formally:

Construction 2.1:

Case 1 ($PN_1 \neq PN_2$)

Let $PN_1 = \langle P_1, T_1, A_1, M_o^1 \rangle$

$PN_2 = \langle P_2, T_2, A_2, M_o^2 \rangle$

And let the result of the joining be

$PN = \langle P, T, A, M_o \rangle$

Then

$$P = P_1 \cup P_2 - \{p,q\}$$

$$T = T_1 \cup T_2 \cup \{\lambda_R, \lambda_A\} - \{R_1, A_1, R_2, A_2\}$$

$$A = A_1 \cup A_2 - \{\langle p, A_1\rangle, \langle R_1, p\rangle, \langle q, R_2\rangle, \langle A_2, q\rangle\}$$

if R_1, R_2 are renamed λ_R and A_1, A_2 are renamed λ_A.

$$M_o(p) = \begin{cases} M_o^1 \ \forall\ p \in P_1 \cap P \\ M_o^2 \ \forall\ p \in P_2 \cap P \end{cases}$$

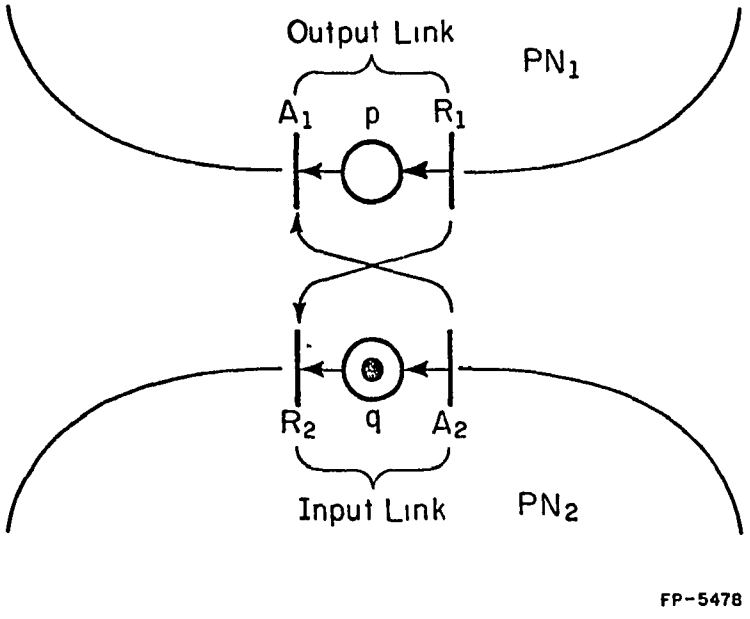


Figure 2.13. Joining Two Networks.

Case 2 ($PN_1 = PN_2$):

$$P = P_1 - \{p,q\}$$

$$T = T_1 \cup \{\lambda_R, \lambda_A\} - \{R_1, A_1, R_2, A_2\}$$

R_1 and R_2 are both in T_1, as are A_1 and A_2.

$$A = A_1 - \{\langle p, A_1\rangle, \langle R_1, p\rangle, \langle q, R_2\rangle, \langle A_2, q\rangle\}$$

if R_1, R_2 are renamed λ_R and A_1, A_2 are renamed λ_A.

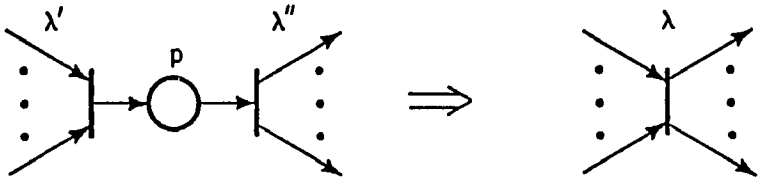
$$M_o(p) = \begin{cases} M_o' \ \forall\ p \neq q \\ 0 \text{ else} \end{cases}$$

The λ transitions are called internal transitions, since they do not correspond to a signal entering or leaving the network of modules. With respect to the external behavior of a network of modules, the firing of this type of transition can be ignored. (Although the transition itself may be necessary, for coordination purposes, to ensure that the correct sequences of signals are modelled.) Hence the following two sequences of transitions associated with PN graphs for networks of modules are regarded as equivalent, from a behavioral point of view:

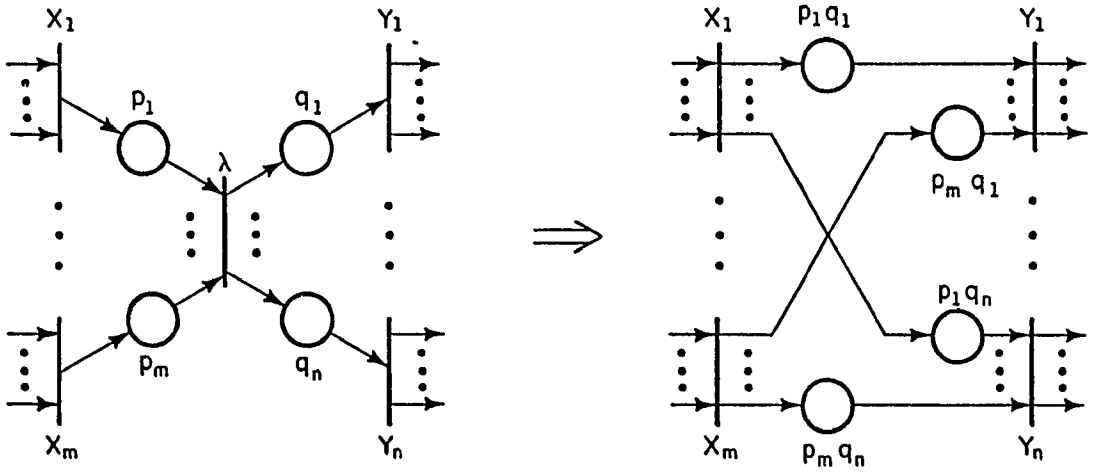
$$R_1\ R_2\ \lambda_3\ \lambda_4\ A_5\ \lambda_6\ R_7$$

$$R_1\ R_2\ A_5\ R_7$$

This leads to the two simplifications shown in Figure 2.14. Applying either of these simplifications to a PN does not alter the behavior that it models. The one at the top of the figure is straightforward: the place p and its input and output arcs are removed, then transitions λ' and λ'' are consolidated into a new transition λ. The input places to λ are those which went to λ' and the output places of λ are those which were fed by λ''. This simplification can be applied only when the only elements of A in which p occurs are $\langle\lambda', p\rangle$ and $\langle p, \lambda''\rangle$. The simplification at the bottom of the figure is a little more complicated, and can be expressed in a more formal



Simplification 1



Simplification 2

FP-5479

Figure 2.14. Two Simplifications.

way as:

Remove from A	Replace with
$\langle X_1, p_1\rangle, \langle p_1, \lambda\rangle$	$\langle X_1, p_1q_1\rangle, \langle p_1q_1, Y_1\rangle$
. . .	. . .
$\langle X_m, p_m\rangle, \langle p_m, \lambda\rangle$	$\langle X_1, p_1q_n\rangle, \langle p_1q_n, Y_n\rangle$
$\langle \lambda, q_1\rangle, \langle q_1, Y_1\rangle$	. . .
. . .	$\langle X_m, p_mq_1\rangle, \langle p_mq_1, Y_1\rangle$
$\langle \lambda, q_n\rangle, \langle q_n, Y_n\rangle$	. . .
	$\langle X_m, p_mq_n\rangle, \langle p_mq_n, Y_n\rangle$

Remove $\{\lambda\}$ from T.

Remove $\{p_1,\ldots,p_m,q_1,\ldots,q_n\}$ from P and replace with $\{p_1q_1, p_1q_2,\ldots,p_mq_n\}$.

If $M_o(p_i) = 1 \Rightarrow M_o(p_iq_1),\ldots,M_o(p_iq_n) = 1$.

If $M_o(q_i) = 1 \Rightarrow M_o(p_1q_i),\ldots,M_o(p_mq_1) = 1$.

(Simplification 1 is just a particular case of Simplification 2.)

Both simplifications will be used in future sections to facilitate arguments concerning the behavioral equivalence of networks of CS modules.

At this point it should be implicitly clear that our view of behavior is one that equates the behavior of a network of CS modules with the sequence of signals into and out of the network. This view has been explored in depth in [Pet 73], where the properties of sequences modelled by PNs are studied.

Figures 2.15 and 2.16 illustrate Construction 2.1. Figure 2.15 shows case 1 and Figure 2.16 case 2. In Figure 2.15 two PNs representing W modules are joined to form a three-output W module. At the top of the figure the unsimplified result of the construction algorithm is shown,

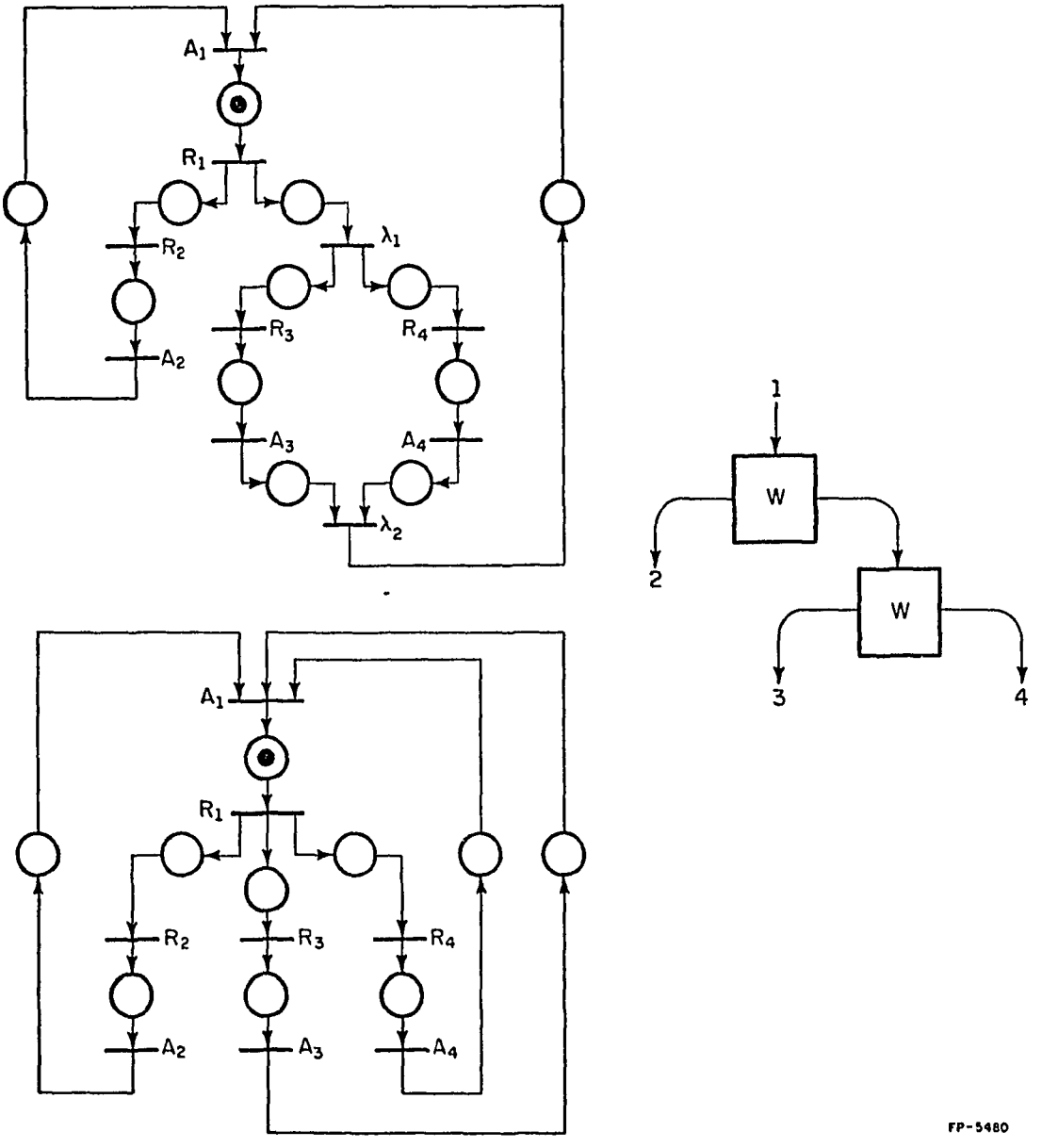


Figure 2.15. Construction 2.1 (case 1).

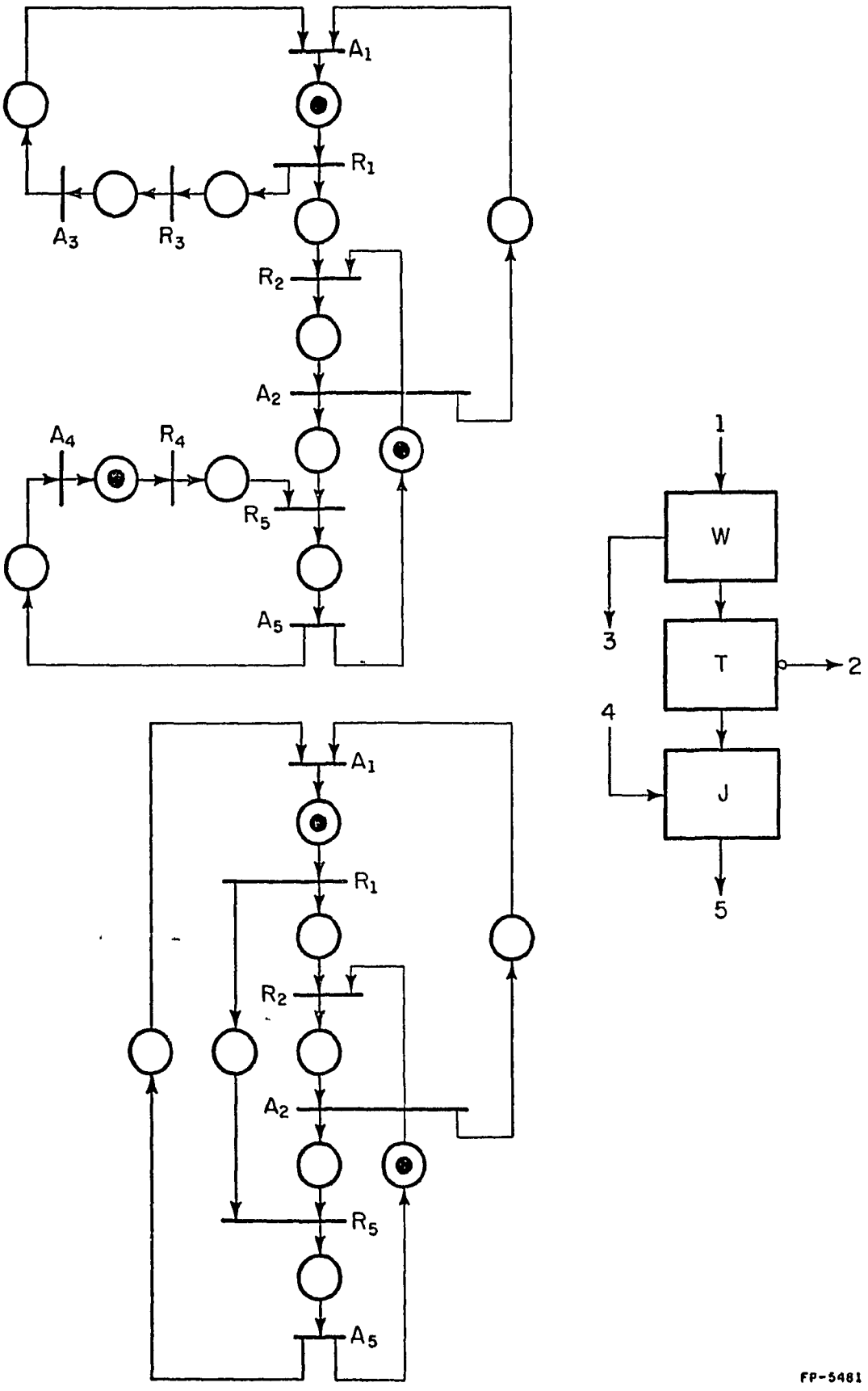


Figure 2.16. Construction 2.1 (case 2).

and at the bottom the simplified result (apply Simplification 2 with m=1, n=2 to λ_1, then with m=2, n=1 to λ_2). In Figure 2.16 the output link 3 of a network consisting of a W module, a T module and a J module is joined to the input link 4. At the top of the figure the PN for the network, before the links are joined, is shown. At the bottom of the figure the PN for the networks, after the links are joined, is shown. The behavior obtained by simulating this PN is equivalent to that obtained from simulating the PN that defines the behavior of the S module (see Figure 2.6). This leads to two observations. One, of incidental interest, that we can construct an S module from a W, a T and a J module. The other, and more important, that the PNs used to define the behaviors of networks of modules need not have equivalent graphs to represent the same behavior. In other words there is no unique PN graph associated with a particular behavior. (However, there is a unique behavior associated with each network of modules.) This deficiency could be rectified by following the ideas presented in [Jum 73]. Jump calls the PN graphs that are used in this discussion "signal graphs". He presents an algorithm for deriving another PN graph, called the "behavior graph", from a signal graph, which is unique for a particular behavior. Although Jump confines his discussion to line and safe marked graphs (a subset of the class of live and safe PNs), his ideas could readily be extended to the class of all live and safe PNs. Since our discussion will not go into very complicated arguments concerning behavioral equivalence, this development, together with the additional formalism, is unjustified.

One final point concerns liveness and safeness. Construction 2.1 case 1 will always result in a live and safe PN, if both component PNs were live and safe to begin with. However, Construction 2.1 case 2 can

result in a PN which is not live and safe even though the original PN was live and safe to begin with (consider a J module with its output and one of its inputs connected).

2.13 Comments on the Modules

The modules presented here are now new. In embryonic form, many of them can be found in [Mul 63]. All of them can be found in [Den 70], with the exception of the SR module. (Even this can be formed from the A module and the U module presented therein.) The SR module can be found in [Pet 74]. Further literature discussing the properties of some subset or another of the modules also includes [Alt 69], [Alt 70], [Bru 71] and [Pat 72]. Other sets of CS modules exist [Bel 72], [Cla 67] and [Kel 74]. Our set was chosen because its members have a natural correspondence with the CHDL.

3. THE SYNTAX OF THE CHDL

Programs in the CHDL define digital systems by describing networks of the CS modules presented in the previous chapter and the register-transfers they control. Before going on to show how programs in the CHDL relate to networks of these CS modules, we shall use this chapter to present the syntax of the CHDL together with some terms that will be useful in later discussions when referring to objects in the syntax.

Figure 3.1 gives the syntax of the CHDL in BNF (Backus-Naur form). Non-terminal symbols are written as sequences of upper case letters. Terminal symbols are underlined sequences of upper and lower case letters, and special characters (brackets, commas, etc.). The terminals are listed in Table 3.1 together with their subsequent representation, if it differs from that shown in Figure 3.1. The following symbols also appear in Figure 3.1.

$$:: = \mid \{ \}$$

They are meta-symbols belonging to the BNF formalism, and not symbols of the CHDL. The first two should be familiar, and the curly brackets denote possible repetition of the enclosed symbols one or more times. In general

$$\{A\} \triangleq A|AA|AAA|...$$

By examining the productions it can be seen that a program (represented by non-terminal PROGRAM) is a list of blocks (BLOCK) terminated by <u>End</u>. The blocks are blocks of statements (STAT, see production 10), and each is headed by an identifier (ID). This is an alphanumeric string unique to the block*.

---

*Not all these stipulations are specified by the syntax of Figure 3.1 alone. These additional syntactic requirements are discussed in Chapter 7.

1.	PROGRAM	:: =	{BLOCK} Dl End
2.	BLOCK	:: =	Dl ID BLOCKBODY
3.	BLOCKBODY	:: =	PROC | DPROC | MPROC | TPROC | WPROC
4.	PROC	:: =	{STAT FIELD3}
5.	DPROC	:: =	Dl Decode (DREG) as DLIST
6.	MPROC	:: =	Dl Mutex {(LABEL, LABEL)} {STAT}
7.	TPROC	:: =	Dl Trigger STAT STAT
8.	WPROC	:: =	Dl While (DREG) do PROC
9.	DLIST	:: =	Dl None ⇒ FIELD2 |{Dl BITS ⇒ FIELD2}|
			{Dl BITS ⇒ FIELD2} Dl None ⇒ FIELD2
10.	STAT	:: =	FIELD1 FIELD2
11.	FIELD1	:: =	Dl LABEL)
12.	FIELD2	:: =	ID|ID [LABEL]|REG-TRF|Null|Wait (DREG)
13.	FIELD3	:: =	# | (ORDER-INFO)
14.	ORDER-INFO	:: =	LABEL|LABEL, ORDER-INFO
15.	LABEL	:: =	{Digit}
16.	ID	:: =	{Letter|Digit}
17.	REG-TRF	:: =	ID ← DREG
18.	BITS	:: =	{0|1}
19.	DREG	:: =	APL expression with IDs as variables.

Figure 3.1. The CHDL Syntax.

There are five types of blocks (see production 3 Figure 3.1): the process block (PROC), the decode process block (DPROC), the mutual exclusion process block (MPROC), the trigger process block (TPROC), and the while process block (WPROC). These are distinguished from each other by the terminal symbol appearing after the block ID (see productions 4 through 8 Figure 3.1).

A PROC block is composed of a list of statements each having three fields (FIELD1 through FIELD3). In the first field there is a numeric label (LABEL), unique within the block to that statement.* In the second field there is either the ID of another block, a register-transfer process descriptor (REG-TRF), a null process descriptor (Null) or a waiting process descriptor (Wait (DREG)). If an ID occurs that is the ID of an MPROC block, it is followed by a numeric label in square brackets. This label must also correspond to a label in the MPROC block of statements referred to.* The third field is optional, and it can contain an n-tuple (any $n > 0$), which should only contain labels from FIELD1 of other statements in the block.*

A DPROC block is distinguished by the terminal symbol Decode following the block ID. (they are separated by a carriage return and line feed). After this comes (DREG), the decode argument, and another terminal symbol as. The remainder of the block is a list (DLIST, production 9) of statements whose first part is either the terminal symbol None or a string of bits, and whose second part is the same as the FIELD2 of a PROC block. The two parts are separated by the terminal symbol ⇒.

An MPROC block is distinguished by the terminal symbol Mutex following the block ID. After this symbol comes a list of pairs, called

*See previous footnote.

Terminals	Representation if different from that in Figure 3.1
Dℓ	Carriage return, line feed (non-printing)
End	
Decode	
(	
)	
as	
Mutex	
,	
Trigger	
While	
do	
None	
⇒	
[	
]	
Null	
Wait	
#	Empty string
Digit	0|1|2|3|4|5|6|7|8|9
Letter	A | • • • | Z
←	
0	
1	

Table 3.1. The Terminal Symbols.

the mutual exclusion condition. These pairs have labels from FIELD1 of subsequent statements in the block as their elements.* The remainder of the block is a list of statements similar to those in PROC blocks except FIELD3 is not present.

A TPROC block is distinguished by the terminal symbol Trigger following the block ID. After this symbol come two statements similar to those in MPROC blocks.

A WPROC block is distinguished by the terminal symbol While following the block ID. After this symbol comes (DREG), called the while argument, and another terminal symbol do. The remainder of the block is similar to a PROC block.

The statements used in the blocks (see productions 9 through 12) are classified as register-transfer types if FIELD2 is a register-transfer process descriptor (REG-TRF), process-call type if FIELD2 is a process-call descriptor (ID|ID [LABEL]), null types if FIELD2 is the null process descriptor Null, and wait types if FIELD2 is the wait process descriptor Wait (DREG) (the (DREG) in this case is the wait argument).

The non-terminal DREG, which occurs in productions 5, 8, 12, and 17, represent an APL [Hil 73] expression with IDs as variables. As we saw in the system model of Chapter 1, these IDs represent registers in the DS. The result of this expression is a vector of bit values. In the case of productions 5 and 17 this vector can be of any length. In the case of productions 8 and 12 it should be only a single bit.

The format of descriptions in the CHDL is controlled by the appearance of the delimiter D1 (carriage return, line feed) in the syntax. Spaces may be included between terminal symbols to aid the readability of the CHDL text.

---

*See previous footnote.

4. INTERPRETING AND TRANSLATING PROGRAMS IN THE CHDL

In this chapter we will present, informally, an interpretation of the CHDL in terms of process behavior, and a procedure for translating programs in the CHDL into networks of CS modules. These two things are related to the following way. The translation procedure associates a program in the CHDL with a unique network of CS modules. Since each network has an unambiguous behavior (see Chapter 2), so does each program. This behavior is our interpretation of the CHDL. Furthermore, this behavior ultimately describes a collection of register-transfer processes (see Chapters 1 and 2), whose functional nature is characterized by the APL expressions in the register-transfer statements of the program. Thus programs in the CHDL unambiguously define digital systems conforming to the system model of Chapter 1.

From the discussion in Chapter 2 it should be clear that, in many cases, a particular behavior may be realized by several different networks of CS modules. Since in our translation procedure we require that each program have a unique modular realization, a choice must be made in specifying the translation procedure. Some choices may result in more efficient realizations than others. This question has been discussed to some extent in [Mud 75] and [Mud 77]. In this discussion we will not consider it.

4.1 The Process Block

Figure 4.1 shows an example PROC block. The CHDL description is shown at the top left. The behavior defined by this block can be obtained by simulating the PN at the top right. This process (called PBLOCK in the CHDL) decomposes into four subordinate processes corresponding to the four

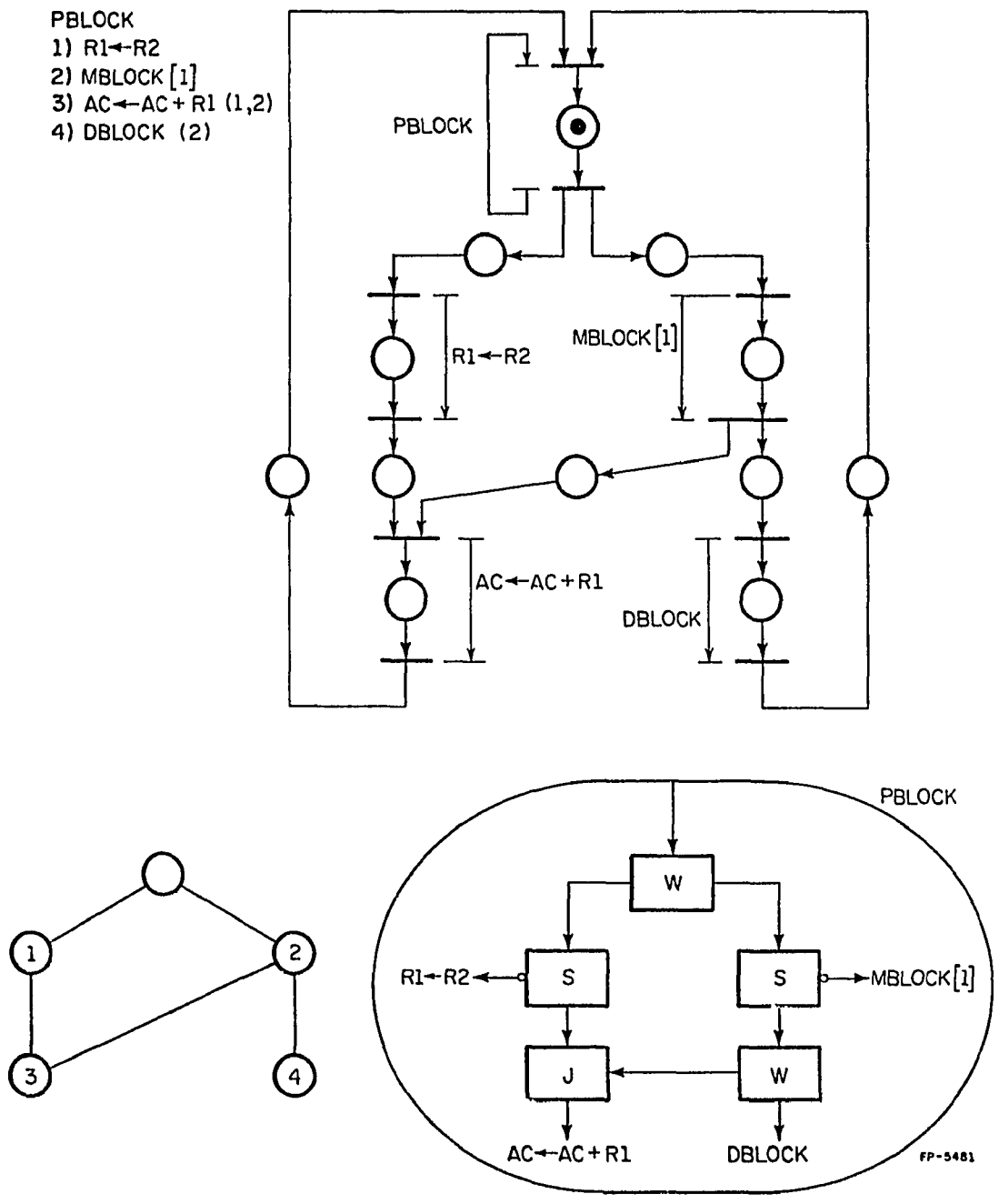


Figure 4.1. An Example Process Block.

statements. The order in which they are to occur is given by the adjacency structure formed from the statement labels and FIELD3 of each statement. This order information can be interpreted as follows. Upon initiating the process PBLOCK, the subordinate processes corresponding to statements 1 and 2 are initiated. When 2 is completed the process corresponding to statement 4 is initiated. When both 1 and 2 are completed, the process corresponding to statement 3 is initiated. The process named PBLOCK is completed when both 3 and 4 are completed. The adjacency structure of PROC blocks can be viewed as a partial ordering of processes in which the underlying binary relation is "precedes in time". These partial orderings each have a universal lower bound (in our example this is the process that controls PBLOCK). The Hasse diagram for the adjacency structure of PBLOCK is shown at the bottom left of Figure 4.1 (we adopt the convention of drawing Hasse diagrams with their lower bound uppermost). The four subordinate processes are as follows: Statement 1 defines a register-transfer: move the contents of register R2 into register R1. Statement 2 defines a process-call, MBLOCK [1]. Statement 3 defines a register-transfer: move the sum of the contents of registers AC and R1 into AC. Statement 4 defines a process-call, DBLOCK.

A register-transfer has already been discussed in the system model of Chapter 1. A process-call is similar to the subroutine construct found in many programming languages. It is a point where the transfer of control to, and the return of control from, another process is made. The behavior of this process is defined by a block whose ID matches the ID used in the process-call. Thus process-call type statements induce a hierarchical ordering on the block structure of a CHDL program. This is a partial ordering that characterizes the control relationship among the blocks. Each block in the program is the controlling process for those blocks that are its immediate

successors in the ordering. Thus in our example, PBLOCK is the controlling process for MBLOCK [1] and DBLOCK. In general, PROC blocks define, by way of their adjacency structures, the temporal relationship among a set of subordinate processes.

The module realization of a PROC block can be derived directly from its Hasse diagram. Each node corresponds to a sub-network of J, S and W modules arranged as follows. The input links to the network are the inputs to a tree of (m-1) J modules. Their output is connected to the input link of an S module. The secondary output link of this S module is connected to the input of a tree of (n-1) W modules. The outputs of the W tree are the output links of the sub-network. The value of m is equal to the number of immediate predecessors of the node, and the value of n is equal to the number of the immediate successors of the node. If two nodes are joined by an arc in the Hasse diagram, an output link of the sub-network corresponding to the "higher" node is connected to an input link of the sub-network corresponding to the "lower" node. In this way the module realization of a PROC block can be systematically constructed. However, two special cases arise in this procedure. Firstly, at least one node has no successor. The sub-networks associated with such nodes are just composed of a tree of (m-1) J modules. Secondly, one node (the universal lower bound) has no predecessor. The sub-network associated with it is a tree of (n-1) W modules.

The important points to notice about the structure of networks formed by this procedure are that they only have a single input link (which is connected to their controlling process, as we shall see in 4.6), that no intra network connections involve the primary output links of S modules, and that each statement (except the Null statement) has a unique output link associated with it. (These links control the processes defined by the

block's statements.) For statements whose associated Hasse diagram nodes have successors, the output links are the primary output links of S modules. For statements whose associated Hasse diagram nodes have no successors, these links may be the output links of W modules, J modules, or the secondary output links of S modules. The Null statement is a special case. It represents the null or empty process. Consequently, it is realized by an Si module. Thus any link associated with a Null statement connects directly to an Si module.

The bottom right of Figure 4.1 shows the module realization of the example PROC block obtained by using the above procedure. By using the PNs of Chapter 2, which define the behavior of the individual modules, together with Construction 2.1 and Simplifications 1 and 2, the reader may verify that the associated PN of this realization is the same as that at the top right of the figure.

4.2 The Decode Process Block

Figure 4.2 shows an example DPROC block. The CHDL description is shown at the top left. The behavior defined by this block can be obtained by simulating the PN in the center of the figure.* This process (called DBLOCK in the CHDL) performs branching based on the value of the two-bit variable X (the decode argument). If X = 00 then control is passed to a process defined by the process-call ABC. If X = 11 then control is passed to a process defined by the register-transfer. All other values of X (signified by the terminal symbol None) result in the null process. In general, DPROC blocks define a multiple way branch process. (There is an obvious analogy with the case statement found in many high level programming languages.)

*In this figure and future ones we shall leave internal transitions unlabelled, unless they are explicitly mentioned in the text.

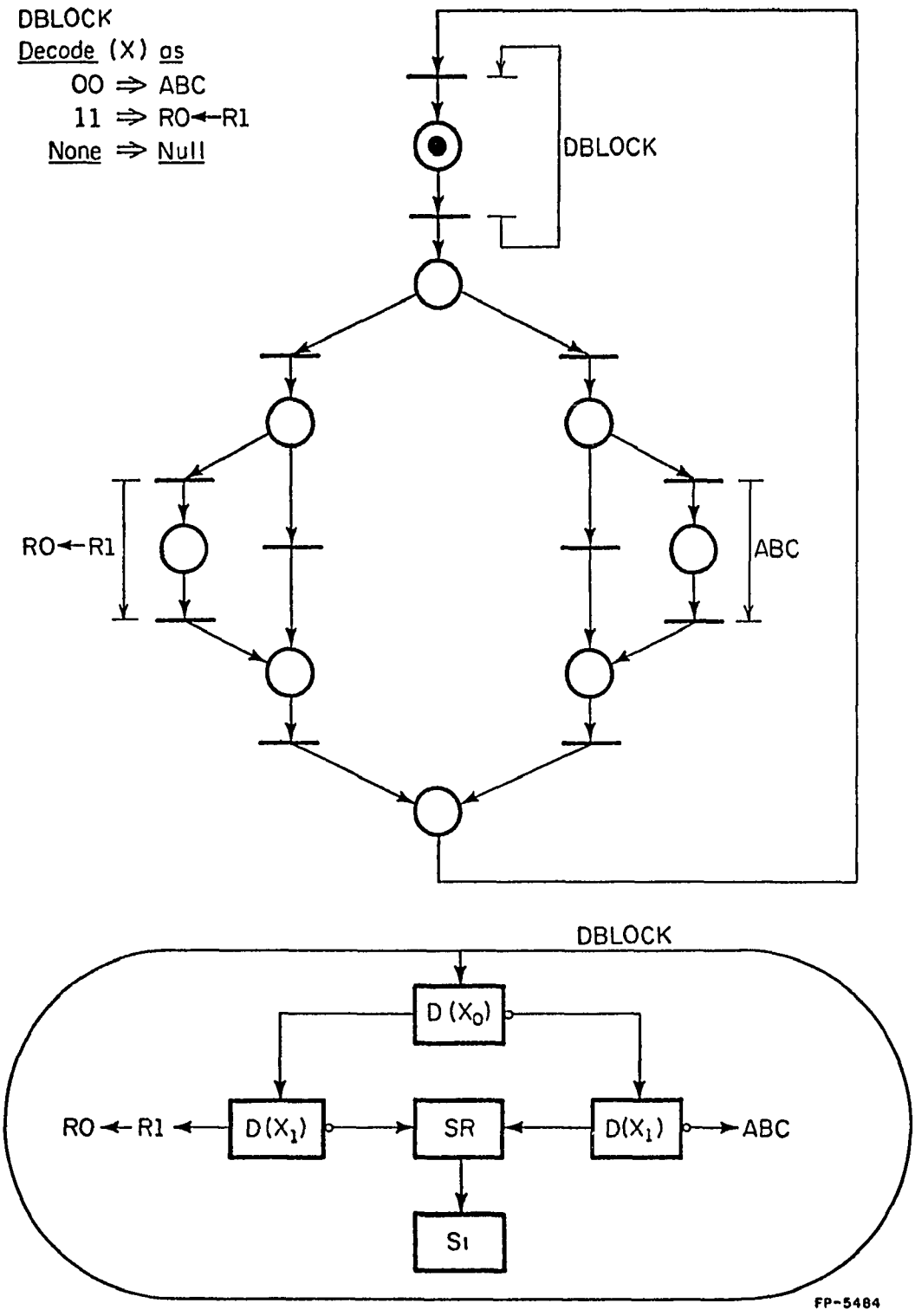


Figure 4.2. An Example Decode Process Block.

The module realization of a DPROC block is simply a tree of D modules. If n is the number of bits in the decode argument, the tree has height n. If those bits are b_1, b_2,...,b_n, then b_i is examined by the conditional links of all D modules at level i in the tree. Thus each D module at level i has b_i as its argument. Those output links corresponding to the decode argument None (i.e. those links in the tree corresponding to none of the explicitly listed values that the argument may take) must access their common process through a tree of SR modules. In our example, this common process is the null process, and the tree is just a single SR module. Strictly speaking, this SR module is an interblock connection, rather than an intra-block connection as shown in Figure 4.2. (See Figure 4.7, section 4.6.).

The bottom of Figure 4.2 shows the module realization of the example DPROC block. Using the methods of Chapter 2 the reader may verify that the associated PN of this realization has the same behavior as the PN at the center of the figure. (Note, however, that the PNs are not the same.)

4.3 The Mutual Exclusion Process Block

Figure 4.3 shows an example MPROC block. The CHDL description is shown at the top. The behavior defined by this block can be obtained by simulating the PN in the center of the figure. This process (called MBLOCK in the CHDL) performs mutual exclusion between certain pairs of three processes that each requires to gain control of a different one of the subordinate processes defined by the block's three statements. These controlling processes are not shown explicitly in the example, but each would contain a process-call type statement whose FIELD2 (see production 12, Figure 3.1) was of the form MBLOCK[i]. The value of i corresponds to the label of the statement defining the subordinate process required by the

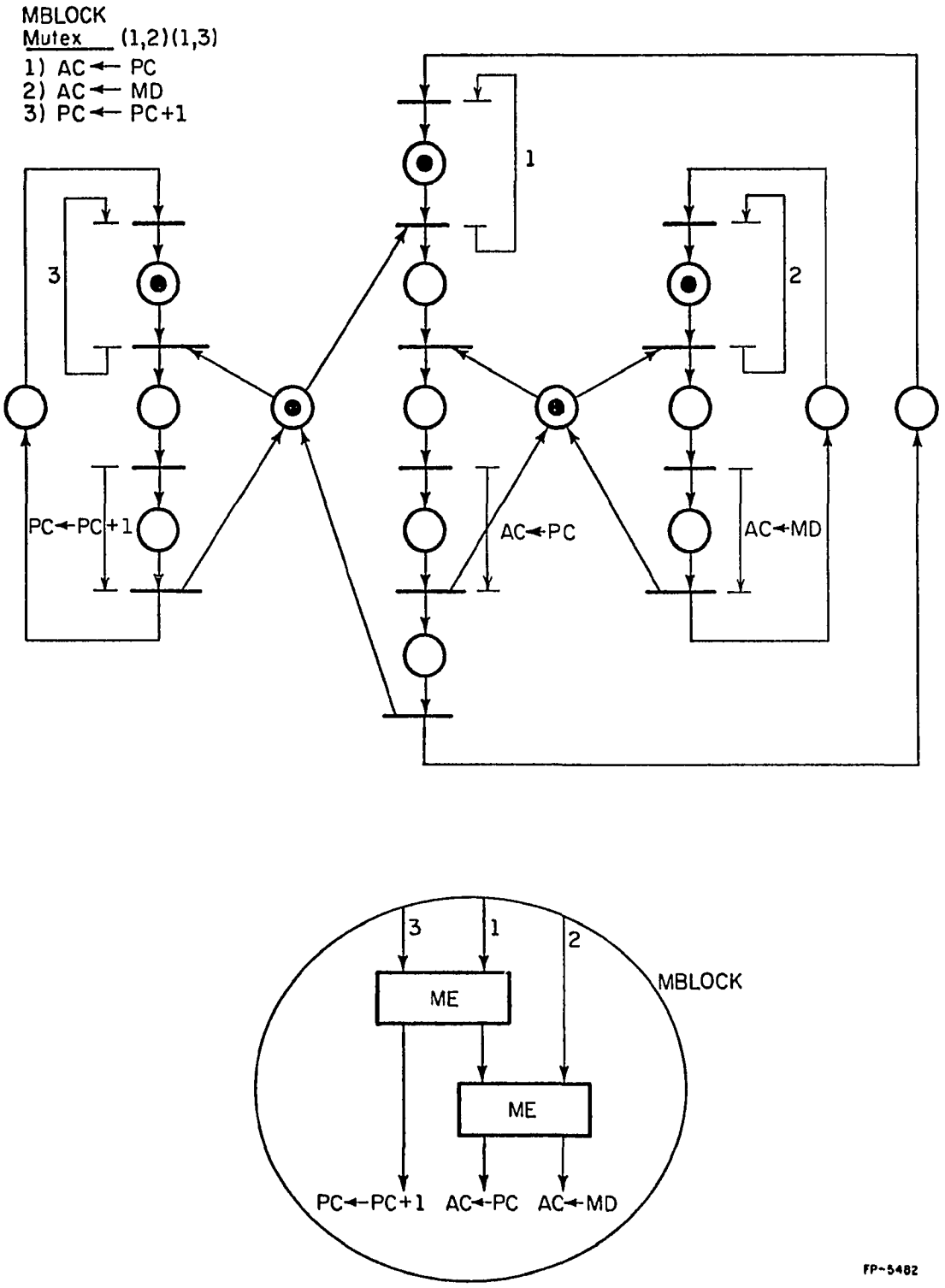


Figure 4.3. An Example Mutual Exclusion Process Block.

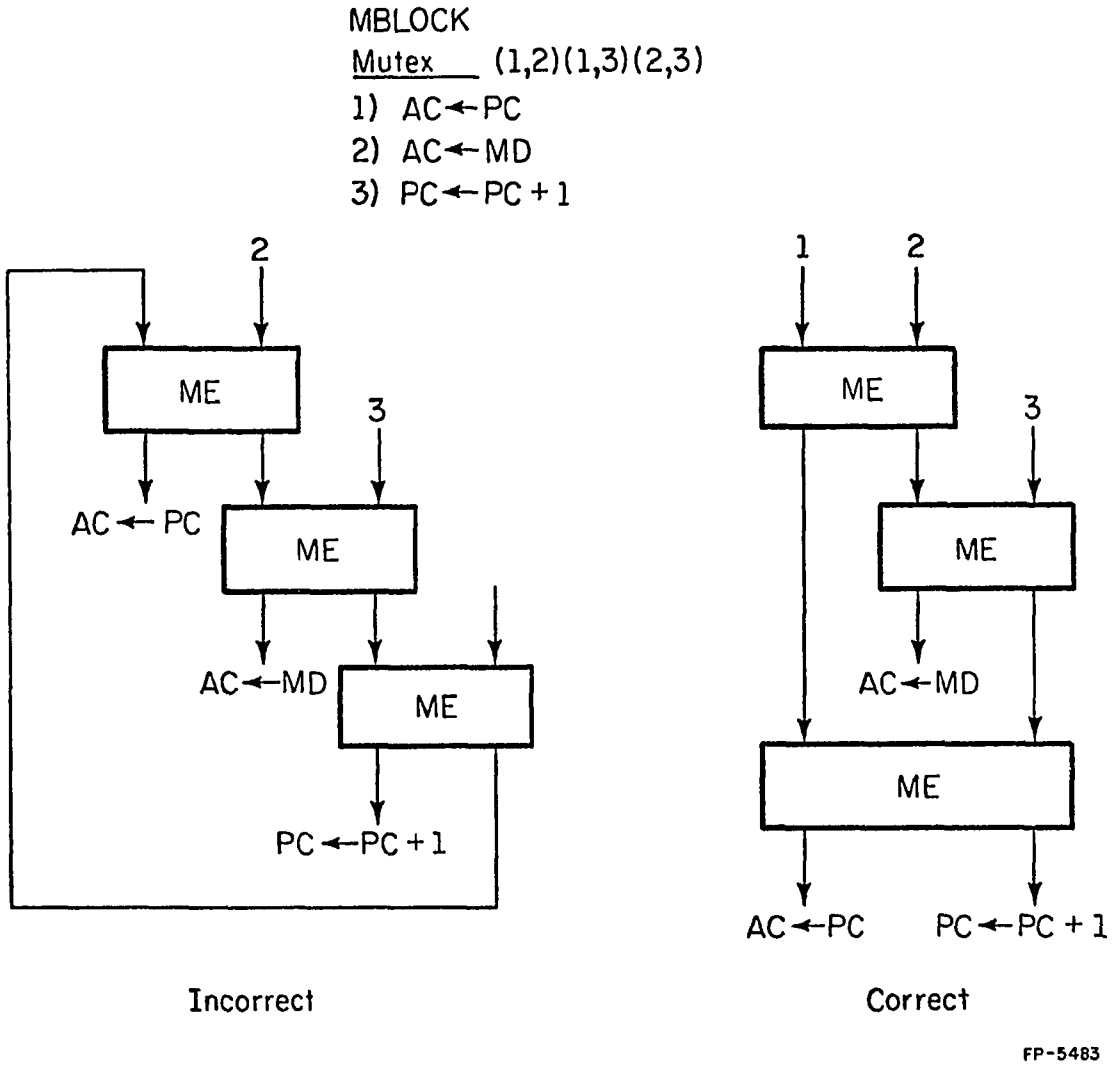


Figure 4.4. Correct and Incorrect Realization of a Mutual Exclusion Process Block.

controlling process. The nature of the mutual exclusion is defined by the mutual exclusion condition (see Chapter 3) following the Mutex symbol. Each pair indicates two subordinate processes, which are to be mutually exclusive in time. Thus in the example, statements 1 and 2 define mutually exclusive processes, as do statements 1 and 3. In general, MPROC blocks define a collection of subordinate processes which are separately controlled, and which would normally be defined within other blocks were it not for the mutual exclusion requirements.

In the example of Figure 4.3 the three statements define register-transfers. The restriction on their concurrency imposed by the mutual exclusion condition ensures that register AC is not being used as the destination of two distinct register-transfers simultaneously, and further, that the register PC is not being modified at the same time as it is being used as the source of a register-transfer.

The realization of any MPROC block can be derived directly from the list of pairs in its mutual exclusion condition. Each one corresponds to an ME module. The rules for their interconnection should be clear from the example in the figure. If the ME modules are regarded as nodes in a directed graph, no realization should contain a circuit. The correct and incorrect way to realize a block in which the potential for this occurs is shown in Figure 4.4. (The block shown here is similar to that in Figure 4.3, except that it has a stronger mutual exclusion condition.) Each statement corresponds to a unique output link and its associated controlling process gains control of the process it defines, through a unique input link. Thus the module realization of an MPROC block has as many input links as output links. (All other block types have only a single input link.)

The bottom of Figure 4.3 shows the module realization of the example MPROC block. Once again the reader may verify that the associated PN of this realization is the same as that in the center of the figure.

4.4 The Trigger Process Block

Figure 4.5 shows an example TPROC block. The CHDL description is shown at the top. The behavior defined by this block can be obtained by simulating the PN shown in the center of the figure. This is the PN for a T module (see Chapter 2, section 2.6). Thus the module realization for this example, as for all TPROCs, is a T module.

The process defined by the CHDL (called TBLOCK) decomposes into two subordinate processes corresponding to the two statements. They both define process-calls, MBLOCK[2] and XYZ. The order in which these subordinate processes are to occur is given by their lexical order of occurrence. Thus MBLOCK[2] precedes XYZ. However, the ability of T modules to control overlapping processes means that the process defined by XYZ may be simultaneously active with the process that controls TBLOCK. In general, TPROC blocks are used to define such overlapping or assembly-line processes.

4.5 The While Process Block

Figure 4.6 shows an example WPROC block. The CHDL description is shown at the top left. The behavior defined by this block can be obtained by simulating the PN at the right. This process (called WBLOCK in the CHDL) is reiterated as long as the while argument is 1. The interpretation and module realization of WPROC blocks is the same as for PROC blocks except for the conditional reiteration. In our example the reiterated part of the process decomposes into three processes. The Hasse diagram corresponding to the order of occurrence of these is shown at the left. The module

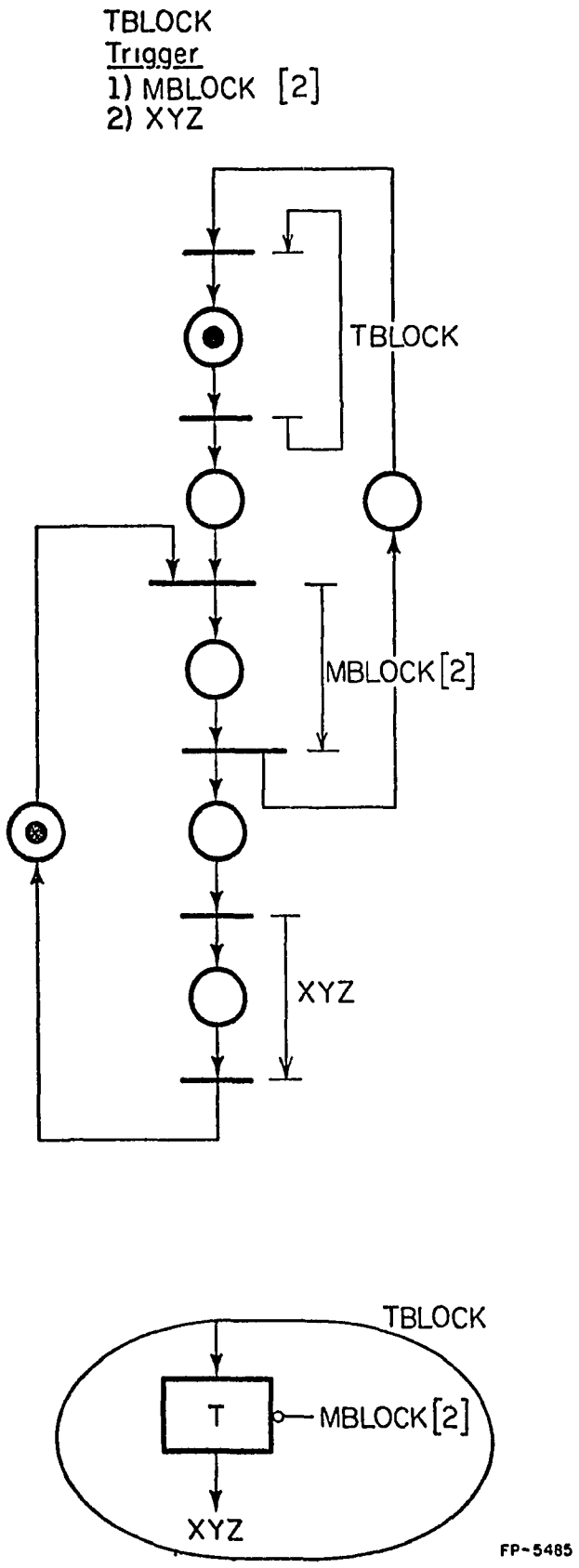


Figure 4.5. An Example Trigger Process Block.

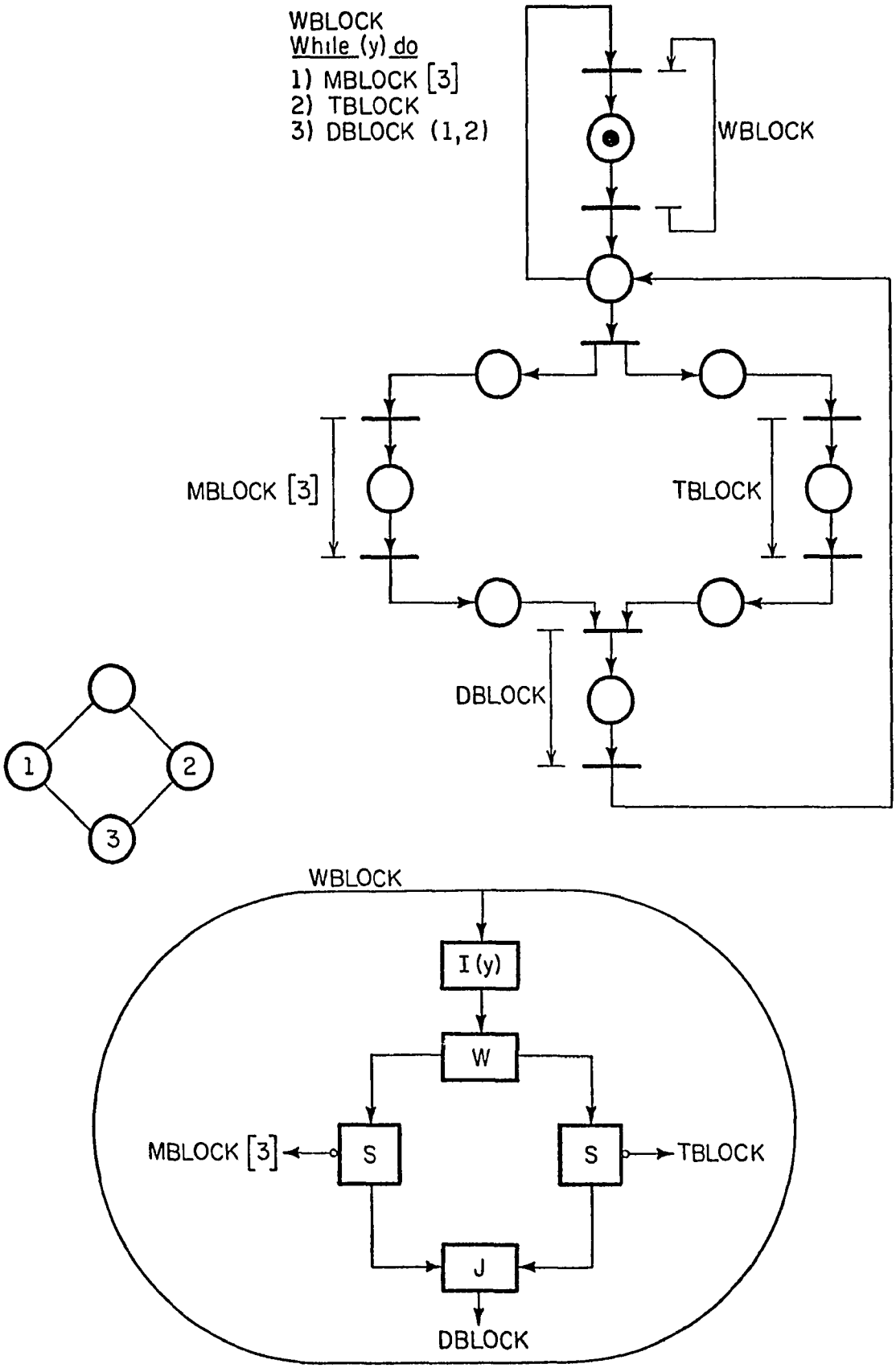


Figure 4.6. An Example While Process Block.

realization of WPROC blocks realizes the conditional reiteration by having an I module at the head of each realization. The argument of the I module corresponds to the 1 bit variable appearing as the while argument.

The module realization of the example WPROC block is shown at the bottom right of Figure 4.6. Once again the reader may verify that the associated PN of this realization is the same as that at the top of the figure.

There is a degenerate form of the WPROC block, and this is the Wait statement. It is realized by connecting an iterate module followed by an Si module to the link associated with the Wait statement. If the 1 bit variable appearing in the wait argument is z, then it may be interpreted as follows: the control process waits at this point as long as z remains 1.

4.6 The Inter Block Connections

Figure 4.7 shows the inter block connections that arise from the five example blocks discussed in the previous sections of this chapter. These are induced by the process-call type statements in the five blocks. Except for MPROC blocks, all block realizations have exactly one input link. If the ID of a process-call statement in some block, A, matched the ID of another block, B, then the link corresponding to the statement in A is connected to the input link of B. If in the whole system of blocks there are n process-call statements with matching IDs, this connection must be done through a tree of (n-1) SR modules. In our example, two SR modules are necessary. One is required, because the process DBLOCK is shared by two other processes (PBLOCK and WBLOCK). The other is required, because the null process is used twice by DBLOCK. The case for MPROC blocks is slightly different, since MPROC block realizations have as many input links as statements. The process-call statements which correspond to the process defined by an MPROC block are of

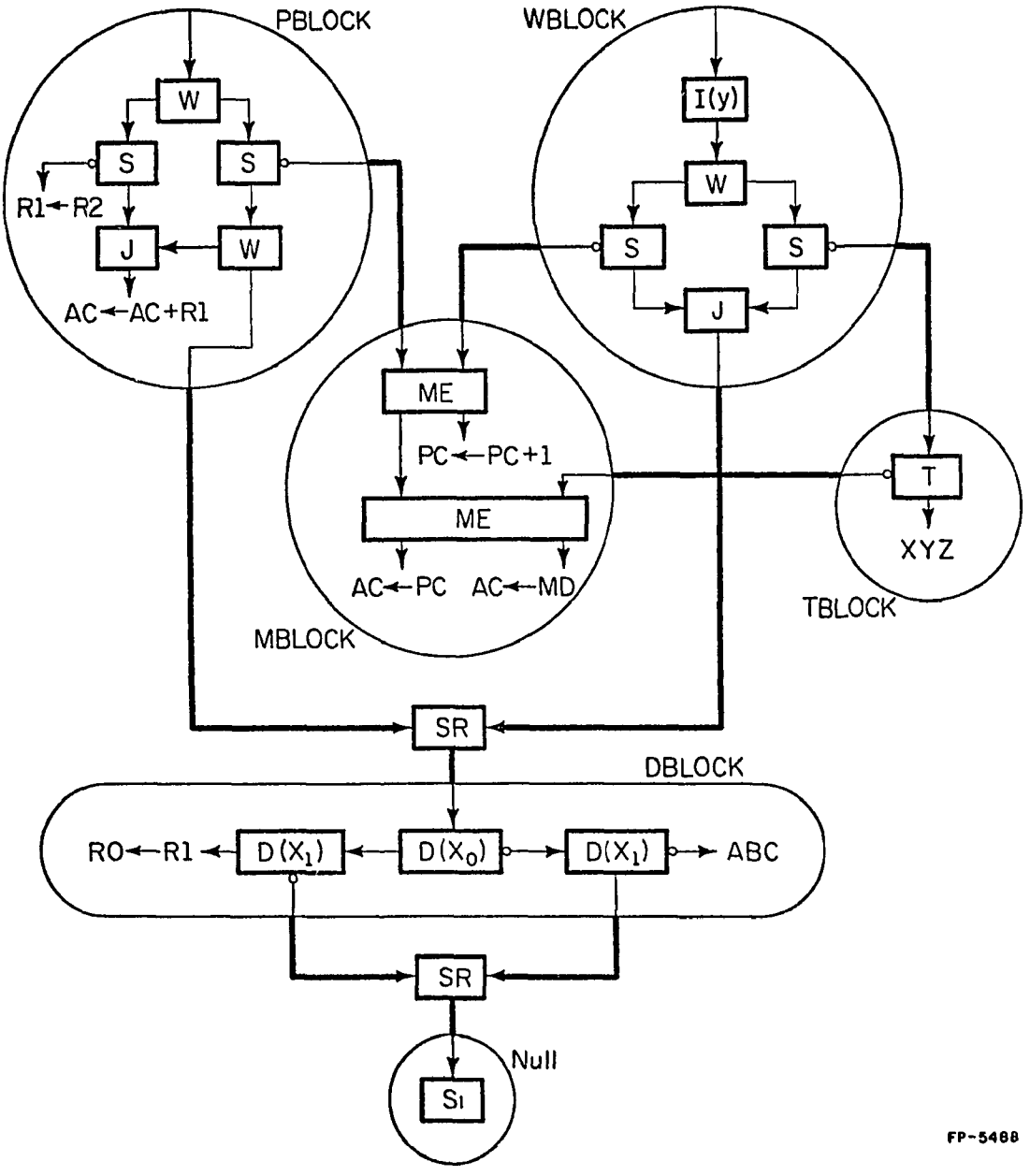


Figure 4.7. Inter Block Connections.

the form ID[i], where ID matches the MPROC block's ID and i matches a statement label in the block. In such cases the connection is made to the input link which corresponds to the statement in the MPROC block having label i. As before, if in the whole system of blocks there are n process-call statements with matching ID[i]s, this connection must be done through a tree of (n-1) SR modules. The input links of those blocks having no predecessors are capped with S_0 modules.

4.7 Comments on the Blocks

The blocks give a convenient way to formulate the design of a system as a hierarchy of less complicated subsystems. Due to the parallelism that the blocks can define, this hierarchical organization can also be viewed as a series of nested partial orderings of processes.

We note also (and we shall see in Chapter 7) that it is partly because the syntax places branch points and mutual exclusion points in special blocks, that CHDL programs only describe CSs that are deadlock-free.

Finally, we note that, because the syntax places T modules in special blocks, CHDL programs show clearly the point at which the CS partitions into overlapping segments. T modules could be placed anywhere S modules are in CHDL defined CSs, without losing freedom from deadlock. However, the operation of such structures would not, in general, be easy to visualize.

5. AN EXAMPLE DESIGN USING THE CHDL

In this chapter the use of the CHDL is illustrated by presenting the design of a small system.

The system is a processor which executes register-to-register instructions. These operate on a DS of four registers and two multi-purpose function units. The CS is implemented as a forwarding algorithm to achieve instruction execution look-ahead.

5.1 The Forwarding Algorithm

Before discussing the CHDL program that describes the design, something should be said about the forwarding algorithm used in the design. It is based on one first presented by Tomasulo [Tom 67], and is aimed at the efficient exploitation of multiple function units. Basic to the technique is a register tagging scheme which permits simultaneous execution of independent instructions while preserving the essential precedences inherent in the instruction stream.

The DS is shown in Figure 5.1. In reality it evolved as the CHDL program was being written. However, for didactic purposes it is convenient to present the completed DS.

The instruction register is shown as IR in Figure 5.1, and the format is two-address register-to-register. The registers (shown as R1 through R4) are specified by fields A and B in IR, and the dyadic operation performed on the data in those registers is determined by the value of the field OC in IR. The instruction is interpreted as follows:

$$RA \leftarrow C(RA) \ \textcircled{OC} \ C(RB) \qquad \begin{matrix} A = 1,\ldots,4 \\ B = 1,\ldots,4 \end{matrix}$$

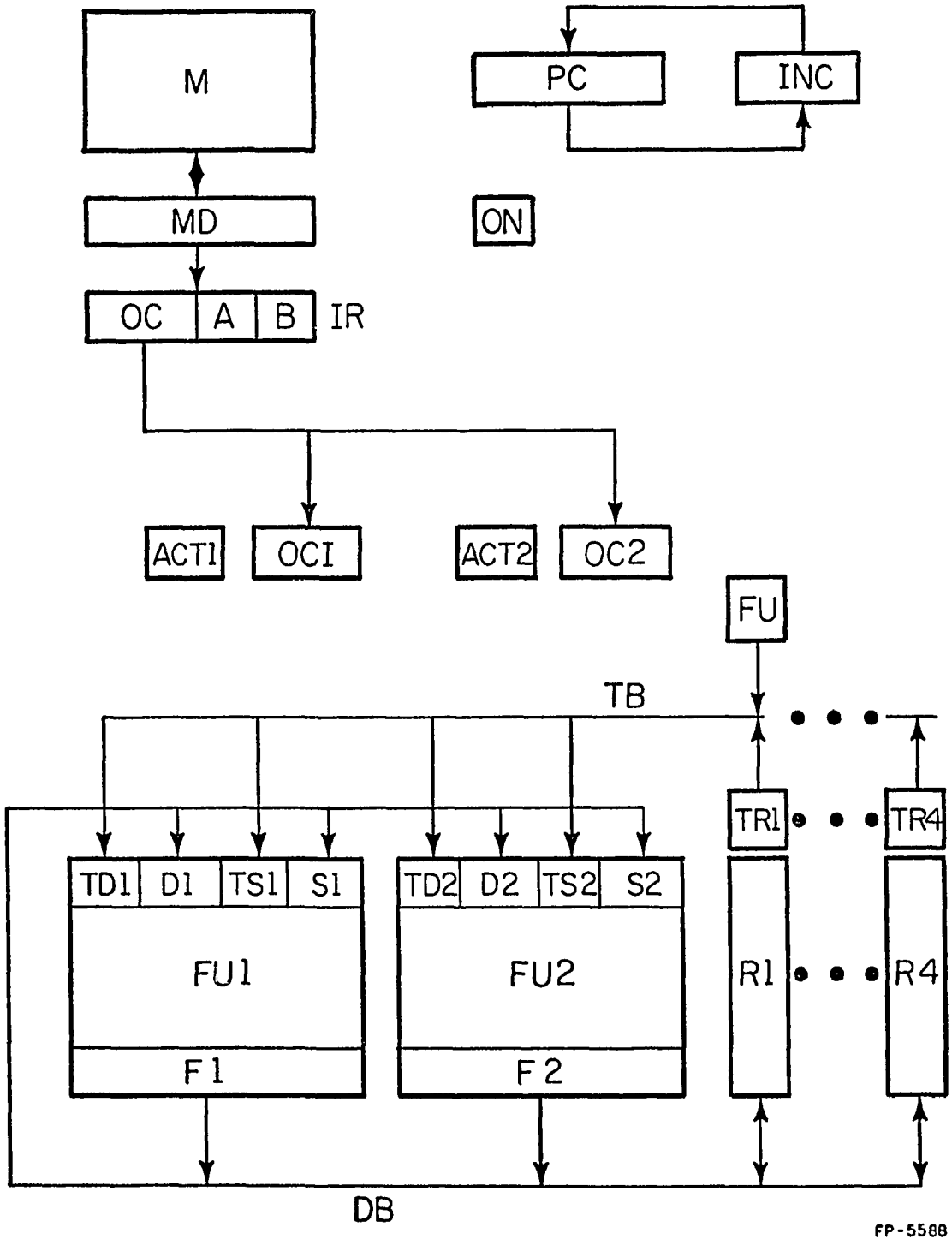


Figure 5.1. The DS of the Example Design.

The operation is performed on one of the multi-purpose function units. The function units are shown as FU1 and FU2, and they perform a variety of (unspecified for this example) dyadic operations. The instruction register IR receives new instructions from the memory data register MD. The MD is the I/O port for the memory M.

The instruction execution breaks down into more basic steps. The contents of the register specified by A is moved over data bus DB to either register D1 or D2, depending on whether FU1 or FU2 has been selected to perform the operation. Next, the contents of the register specified by B is moved over DB to either register S1 or S2 (again depending on which function unit has been selected). The function unit performs the operation taking the contents of registers D1 and S1 (if it is FU1), or D2 and S2 (if it is FU2) as its operands. It deposits the result into register F1 (if it is FU1), or F2 (if it is FU2). This result is then moved over DB to the register specified by A. This basic instruction execution process can undergo some modification as we shall see.

Instructions are issued to IR whenever there are available function units to execute them. However, the registers specified by fields A and B may be being used by previously issued but uncompleted instructions. This is resolved by the forwarding algorithm. As part of this algorithm, tag registers TR1 through TR4, TD1, TS1, TD2 and TS2, each two bits long, are associated with registers R1 through R4, D1, S1, D2 and S2 respectively. In decoding each instruction the DS checks the tag registers of both of the specified registers. If they are both 00 the execution of the instruction can proceed, and the tag register of the register specified by A is set to 01 (if FU1 is to perform the operation specified by the instruction), or 10 (if FU2 is to perform the operation). The non-zero value in the tag

register indicates that the contents of the associated register are in the process of being changed by an instruction execution, and it also indicates from which function unit the new contents are to come. If, when decoding an instruction, either of the tag registers does not contain 00, it indicates to the CS that the associated register(s*) will be used to receive the result of an instruction which was issued earlier, and which is still in the process of being executed. Hence,the current instruction execution must wait for the result of the earlier instruction execution. The issuing and execution of further instructions could be delayed at this point until the required result is in. However, the forwarding algorithm avoids this potential inefficiency by sending the non-zero tag(s*) over bus TB to the appropriate function unit, in lieu of the result, to reserve that unit for when the result is in. Next the tag register of the register specified by A is set to 01 (if FU1 is to perform the operation specified by the instruction), or 10 (if FU2 is to perform the operation). (Note that this may involve over-writing non-zero data in the tag register.) The next instruction can now be issued.

If the A field of an instruction called for the contents of R1 as an operand, and the instruction's operation was to be performed by FU1, and further that the tag register associated with R1, namely TR1, was non-zero, then the contents of TR1 would be moved over TB to TD1, and TR1 would be set to 01. Also, if the register called for by field B of the instruction was in use, the contents of its tag register would be moved to TS1. (TD2 and TS2 would be the tag destinations, if FU2 was the function unit to be used.)

*Since this example has only two function units whose operations are not pipelined (implying that their input registers are not allowed to be changed until their operations are complete and their results have been output), these cases do not occur.

A summary of the interpretations placed on the tag data is given below in Table 5.1.

TRi (i = 1,...,4)	
00	Data available in Ri
01	Ri in use, result expected from FU1
10	Ri in use, result expected from FU2

TD1 or TS1	TD2 or TS2	
00	00	Data in D or S
*	01	Result expected from FU1
10	*	Result expected from FU2

Table 5.1. Tag Data.

In its execution of the dyadic operation specified by OC, function unit i first checks to see if V/(TDi, TSi) = 0. If it does, it interprets this to mean that both operands are present in Di and Si. It can then proceed with the operation. If the above condition is not true (i.e. the function unit has received non-zero tag data instead of operand data), the unit waits until it is. When it completes its operation and places the result in Fi, it broadcasts this result to all registers whose tag registers contain data agreeing with the two bit code associated with that function unit (see Table 5.1). In many cases, therefore, results are "forwarded" straight to the D or S registers of the function units, rather than going

*See previous footnote.

via any of the registers R1 through R4. When a broadcast result reaches a register, the associated tag register is reset to 00. In the case of function units waiting on operands this is the go-ahead to start operation.

This algorithm has the property of preserving essential precedences in the instruction stream, while allowing independent instructions to be executed in an order which is dictated only by the availability of a function unit. When the dyadic operations can take a long time compared to the register-to-register movements, this makes for efficient utilization of the multiple function units.

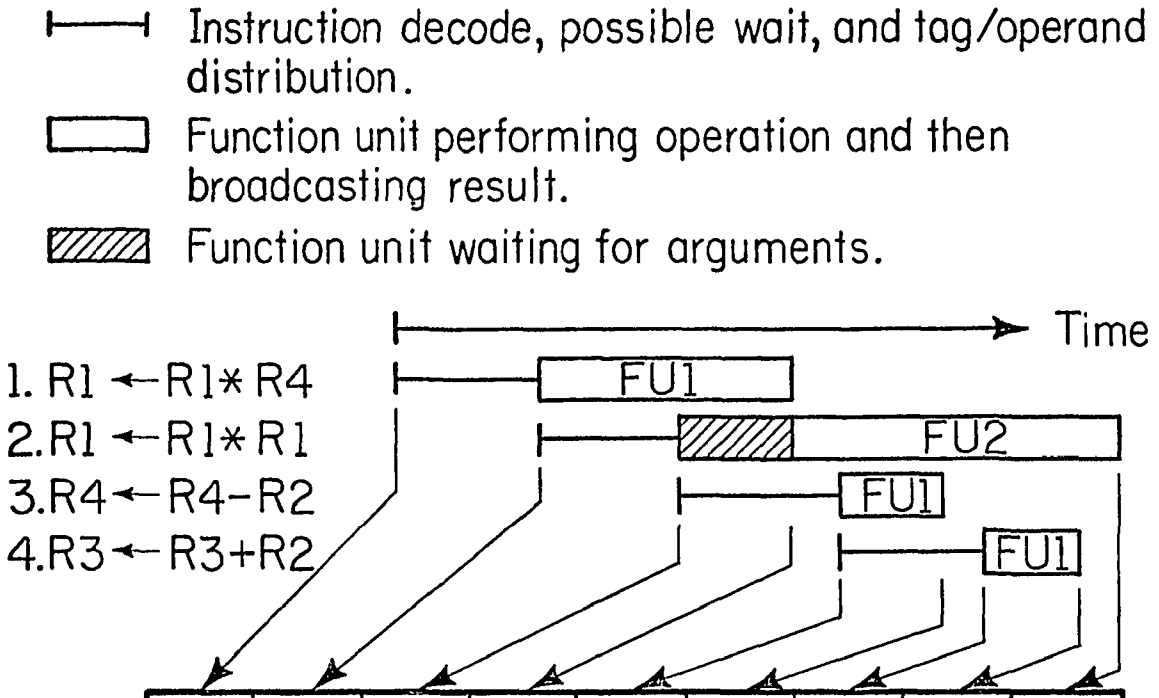
Figure 5.2 illustrates the algorithm in operation on a stream of four instructions. This is for the DS of Figure 5.1. The contents of the registers and their tags are shown at key times in the execution process.

5.2 The CHDL Program for the Example Design

Figure 5.3 shows the CHDL program for the example design. Notice that we have used two "lexical" indices i and j. These take values from the sets {1,2,3,4} and {D,S} respectively. Thus in Figure 5.3(b) we have used RAi to stand for the four blocks RA1, RA2, RA3 and RA4. Furthermore, within each block i is replaced with the appropriate value 1,2,3, or 4. Similarly in Figure 5.3(e) we have used DECTj1 to stand for the two blocks DECTD1 and DECTS1.

Figure 5.4 shows the block dependencies. If one block in the program calls another,* this relationship is represented in Figure 5.4 by a downward sloping line from the calling block to the one called. Precedence between blocks called from the same block is not represented, neither is the block

*Borrowing a term used in software to describe an analogous situation, we say that block A calls B if A has a process-call statement whose ID matches B's ID.



TR1	00	01	10	10	10	10	10	10	00
R1	A	A	A	A	A	A	A	A	A^2D^2
TR2	00	00	00	00	00	00	00	00	00
R2	B	B	B	B	B	B	B	B	B
TR3	00	00	00	00	00	00	01	00	00
R3	C	C	C	C	C	C	C	C+B	C+B
TR4	00	00	00	00	01	00	00	00	00
R4	D	D	D	D	D	D−B	D−B	D−B	D−B
TD1	00	00	00	00	00	00	00	00	00
D1	—	A	A	A	D	D	C	C	C
TS1	00	00	00	00	00	00	00	00	00
S1	—	D	D	D	B	B	B	B	B
TD2	00	00	01	00	00	00	00	00	00
D2	—	—	—	AD	AD	AD	AD	AD	AD
TS2	00	01	01	00	00	00	00	00	00
S2	—	—	—	AD	AD	AD	AD	AD	AD

FP-5589

Figure 5.2. Illustration of the Forwarding Algorithm.

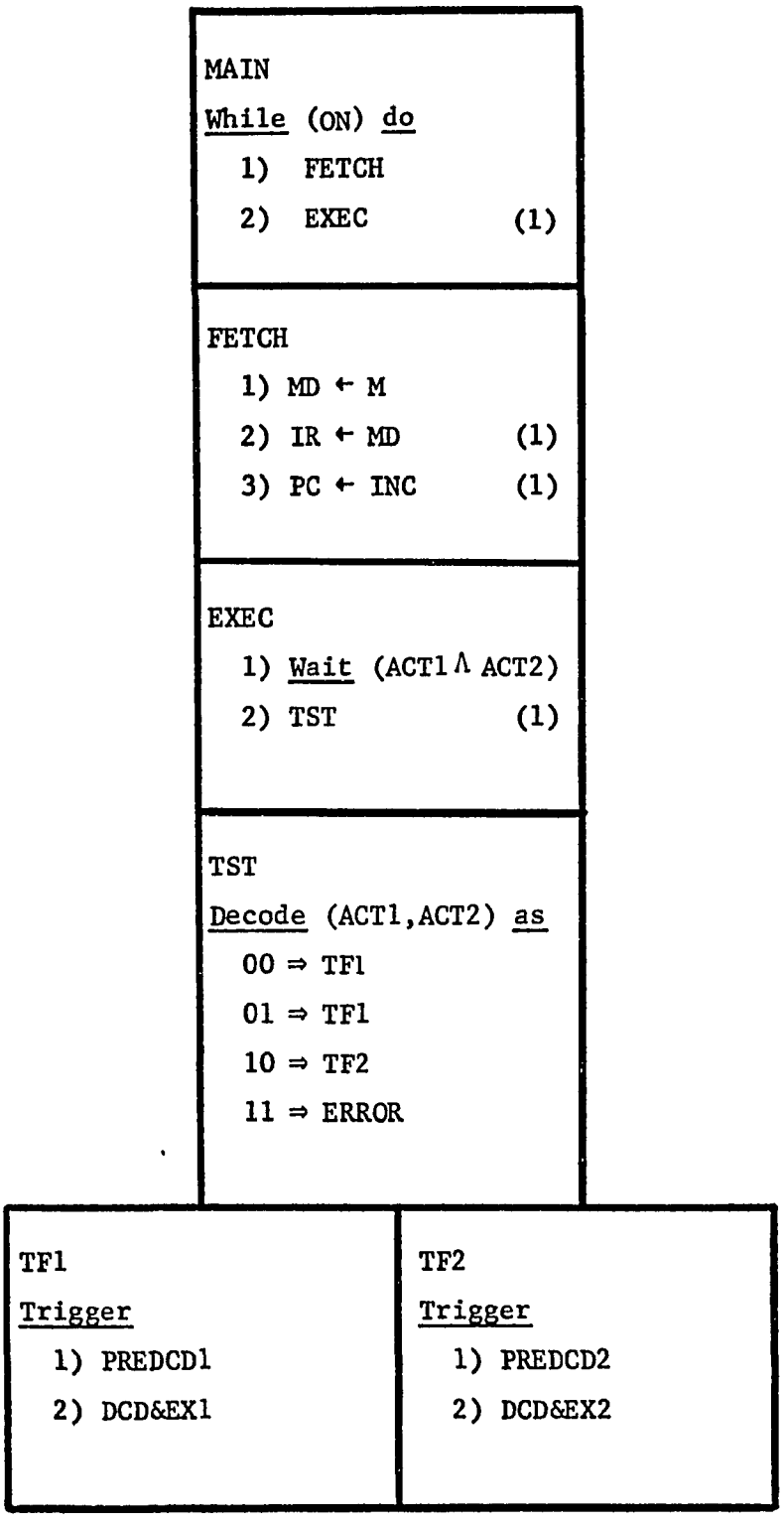


Figure 5.3(a). The Example Design.

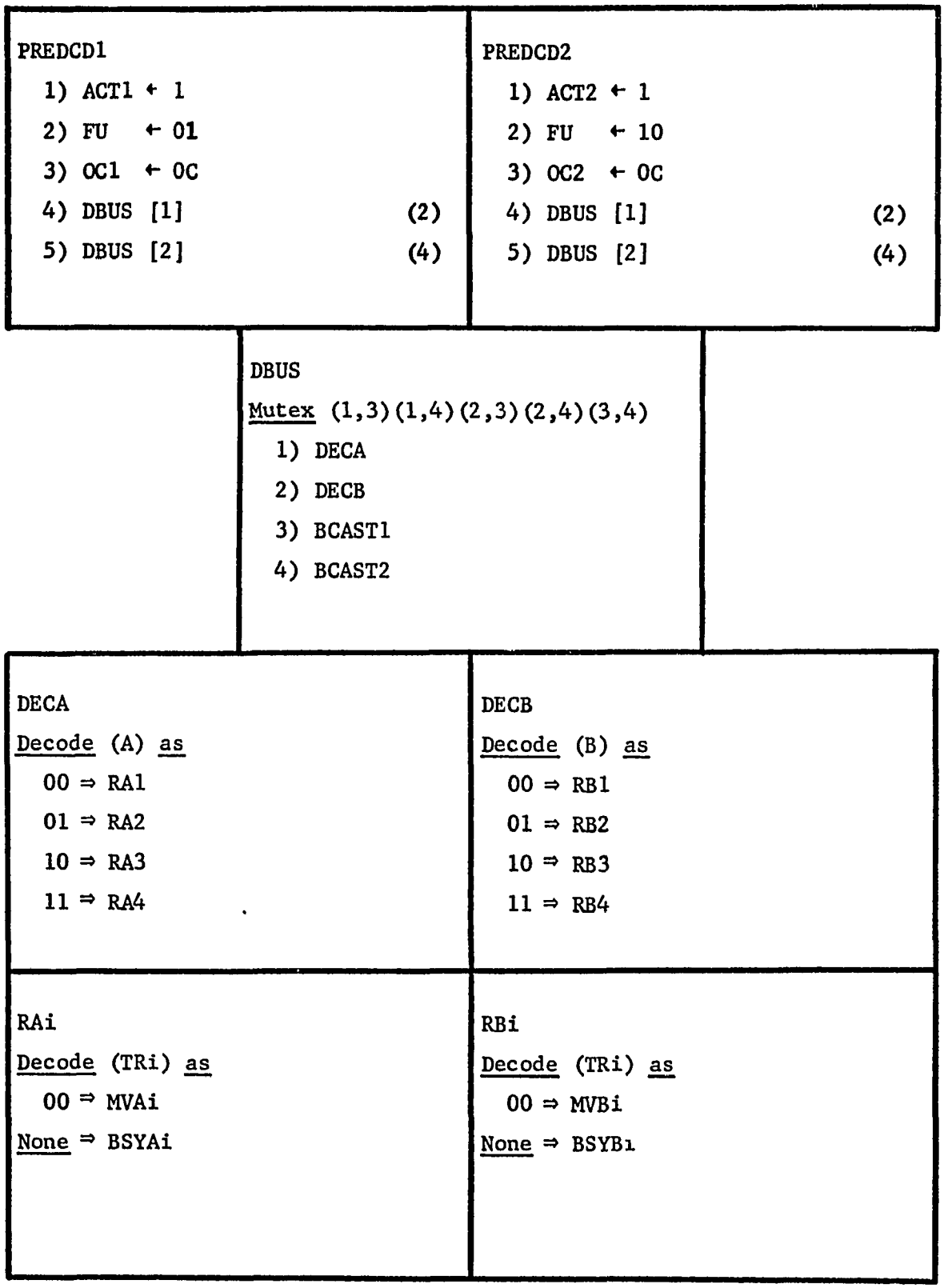


Figure 5.3(b). The Example Design.

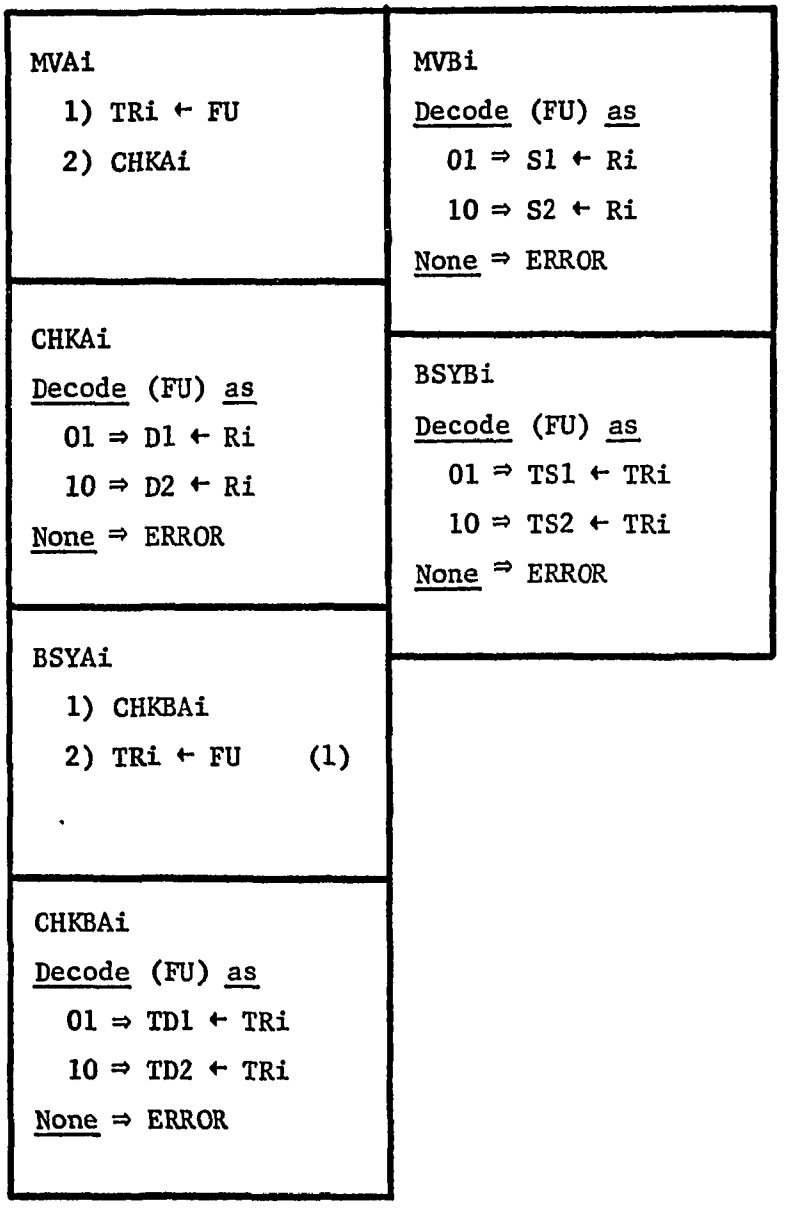


Figure 5.3(c). The Example Design.

DCD&EX1	DCD&EX2
1) Wait (V/(TD1,TS1))	1) Wait (V/(TD2,TS2))
2) F1 ← D1 (OC1) S1 (1)	2) F2 ← D2 (OC2) S2 (1)
3) DBUS [3] (2)	3) DBUS [4] (2)
BCAST1	**BCAST2**
1) DEC1TR1	1) DEC2TR1
2) DEC1TR2	2) DEC2TR2
3) DEC1TR3	3) DEC2TR3
4) DEC1TR4	4) DEC2TR4
5) DECTD2	5) DECTD1
6) DECTS2	6) DECTS1
7) ACT1 ← 0 (1,2,3,4,5,6)	7) ACT2 ← 0 (1,2,3,4,5,6)

Figure 5.3(d). The Example Design.

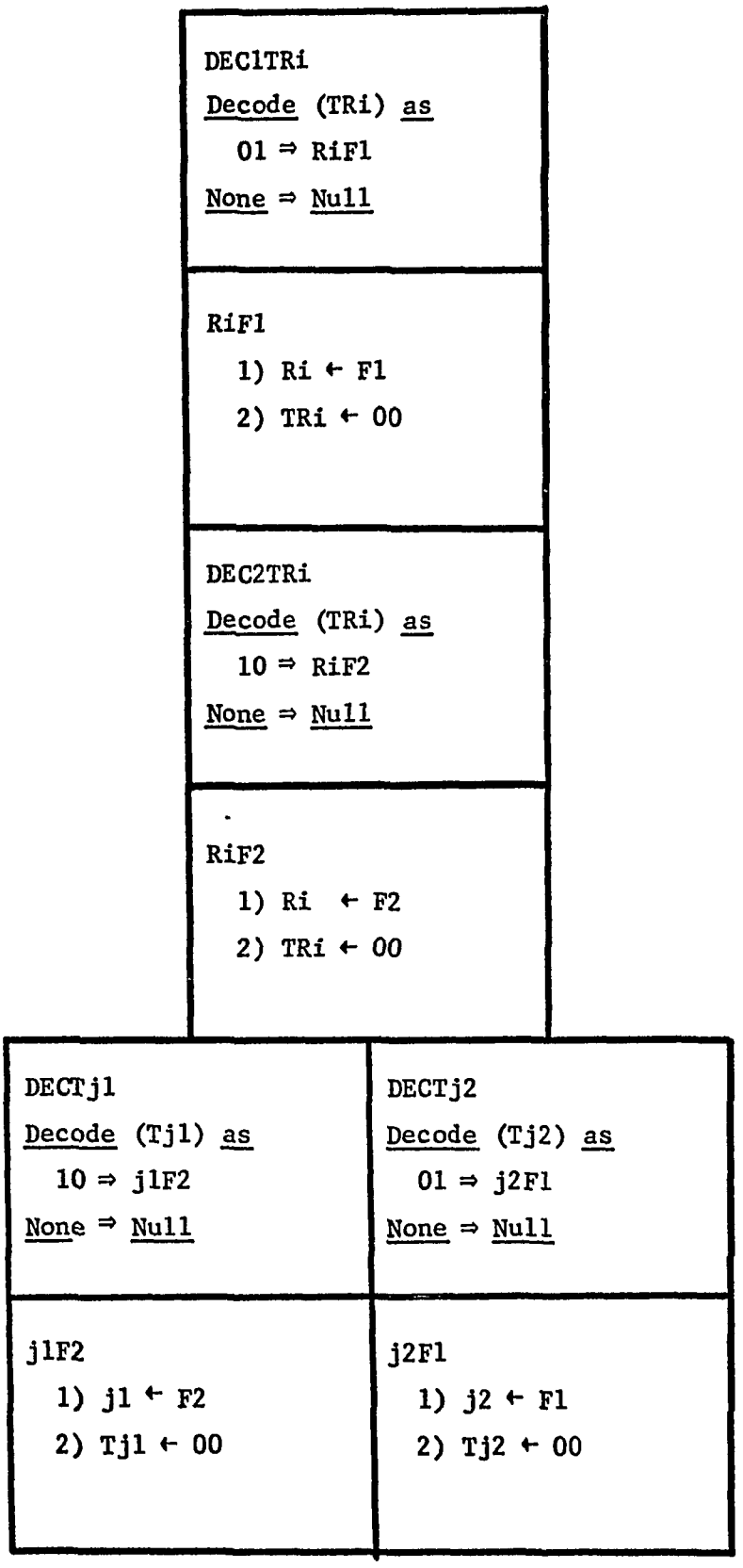


Figure 5.3(e). The Example Design.

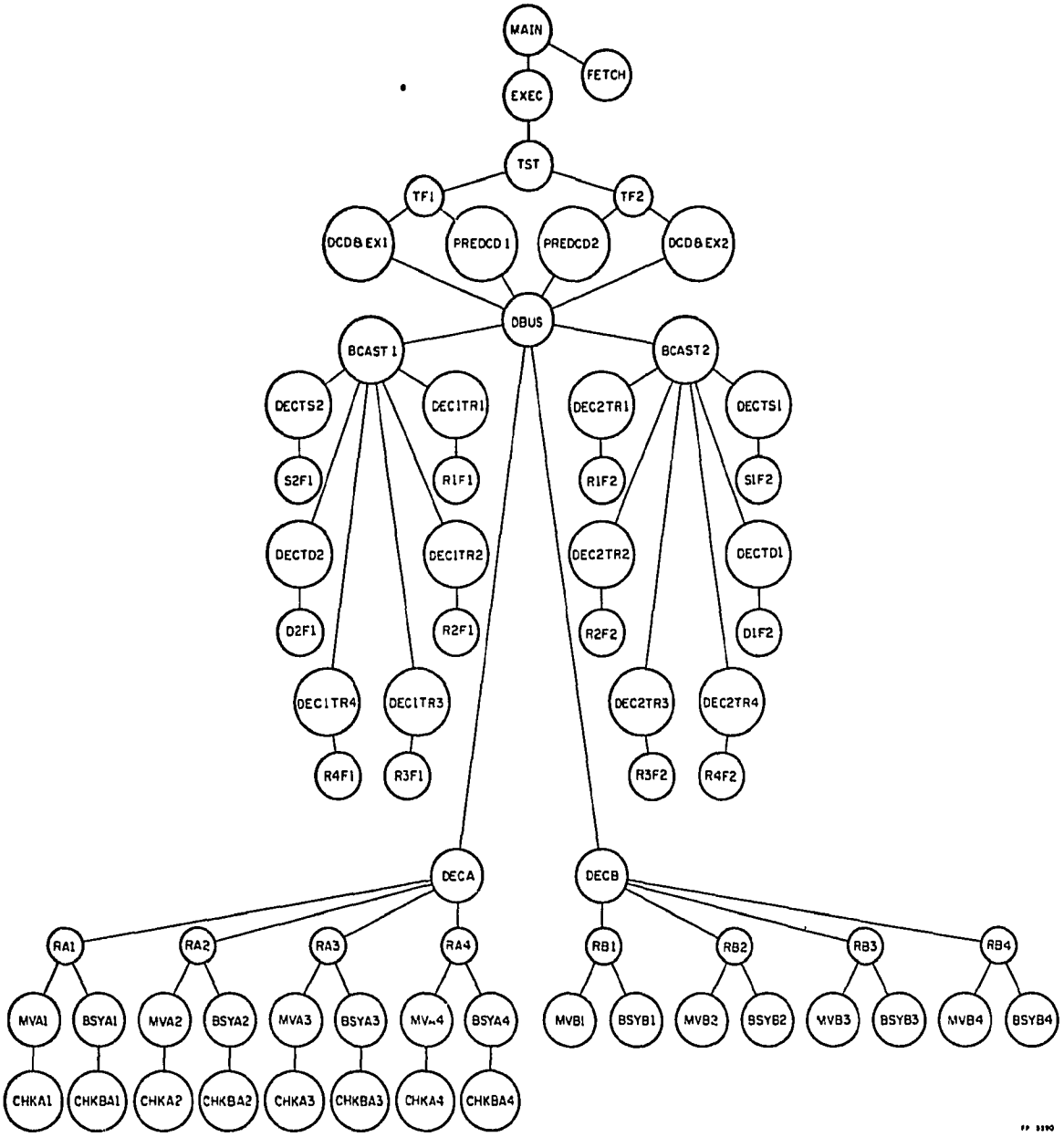


Figure 5.4. The Block Dependencies.

type (i.e. PROC, DPROC, MPROC, etc.). The shorthand of using the lexical indices of Figure 5.3 has been dropped, and each block ID has been written out in full.

5.2.1 The MAIN Block

This is the highest block in the hierarchical structure of blocks (see Figure 5.4). It therefore represents the most simple description of the target system. It partitions the system, which we have called MAIN, into two subsystems, specified by blocks FETCH and EXEC. FETCH is run first, and upon its completion EXEC is run. This sequence is reinitiated as long as the ON flag is set. As may be guessed, the names are mnemonic; FETCH describes a process that fetches an instruction from memory and EXEC describes a process that executes it.

5.2.2 The FETCH Block

This decomposes into three register-transfers. These move the contents of the memory location (M) pointed to by the program counter (PC) (the contents of the memory location is assumed to be an instruction) into the memory data register (MD), and then into the instruction register (IR). Concurrently with this last register-transfer the contents of PC are incremented. (INC is a functional block whose output is its input plus one.)

5.2.3 The EXEC Block

The flags ACT1 and ACT2 are used to indicate to the CS the availability of FU1 and FU2, respectively. If FU1 is busy ACT1 is set, similarly for FU2 and ACT2. Thus the process described by block EXEC waits until at least one of the function units is available before proceeding with the execution of the instruction in IR.

5.2.4 The TST Block

This block is called by block EXEC. It describes a process that assigns the execution of the instruction in IR to either block TF1, if FU1 is available (i.e. ACT1 = 0), or else to block TF2. The choice is based on the two bit value (ACT1, ACT2).

The occurrence of (ACT1, ACT2) = 11 at this point in the operation of the system is clearly an error. This is dealt with by block ERROR, whose details we have not specified.

5.2.5 The Blocks TF1 and TF2

These two blocks describe similar but mutually exclusive (as a consequence of the decode process TST) processes. TFi (i = 1,2) comprises process PREDCDi followed by process DCD&EX1. PREDCDi describes the movement of data and/or tags specified by fields A and B of the instruction in IR, from any of the registers R1, R2, R3 or R4 to function unit i. DCD&EXi describes the execution, by function unit i, of the operation specified by the instruction, and the subsequent broadcasting of the result to the registers and function units.

Since TF1 and TF2 are TPROCs, both DCD&EX1 and DCD&EX2 can overlap with the FETCH process and the first part of the EXEC process up to but not including PREDCD1 and PREDCD2, respectively. Assuming the function units are initially not busy this allows the following type of occurrence: the system can fetch an instruction, issue it to a function unit, fetch a second instruction, issue it to the other function unit, then finally fetch a third instruction. At this point, the possibility of a wait (based on the availability of a function unit) at the beginning of process EXEC determines when this third instruction gets issued to a function unit. Thus the system

keeps two instructions executing concurrently, one in each function unit, and a third ready for issue in IR. The details of this will become clearer as we continue with comments on the program of Figure 5.3.

5.2.6 The Blocks PREDCD1 and PREDCD2

These preliminary decode processes set up some status registers to assist in tag manipulation and function unit operation. They also call other blocks to perform data and/or tag movement. Specifically, in PREDCDi, flag ACTi is set (indicating that FUi is to be busy). Register FU is set to 01 (for i = 1), or 10 (for i = 2). This register is used to indicate which function unit is to perform the instruction's operation, and as a source for tag data. The operation code (in field OC of IR) is sent to the operation register of the function unit (OC1 or OC2). Finally, two decode processes are initiated to determine from which registers the instruction's operands are to come. These decode processes, DECA and DECB, are called sequentially, and they are called through a mutual exclusion process, DBUS.

5.2.7 The Block DBUS

This mutual exclusion process ensures that the movement of data and/or tags from the registers to the function unit, the broadcasting of results from function unit 1, and the broadcasting of results from function unit 2 do not interfere with one another as a result of using a common resource, DB. (BCAST1 and BCAST2 are the blocks that handle the broadcasting. They are discussed later on.)

Notice that PREDCD1 and PREDCD2 are mutually exclusive in time (since TF1 and TF2 are). Thus DBUS and, hence, DECA and DECB can be shared by them without conflict. Also, since the decode processes, DECA and DECB, are called sequentially (see the order information of statements 4 and 5 in

PREDCD1 and PREDCD2) they do not interfer with one another even though some register-transfers that they ultimately invoke share DB. Thus it is not necessary to include the pair (1,2) in the mutual exclusion condition of DBUS.

5.2.8 The Blocks DECA and DECB

The blocks DECA and DECB decode the operand fields A and B respectively, to find out which of the registers, R1, R2, R3 and/or R4, are to be used by the instruction in IR. Based on the result of these decode processes, control is passed to other processes to handle the movement of data and/or tags between the registers and the function units.

For example, in DECA, if A = 01, control passes to block RA2. This moves the contents of R2 (using MVA2) or its tag (using BSYA2) to the designated function unit.

Similarly, for example in DECB, if B = 00, control passes to block RB1. This moves the contents of R1 (using MVB1) or its tag (using BSYB1) to the designated function unit.

5.2.9 The Blocks RA1 and RBi

The blocks RA1,...,RA4, RB1,..., and RB4, describe decode processes. Block RAi (i = 1,...,4) calls blocks MVAi if tag register TR1 = 00 (i.e. if register Ri is not being modified by an instruction execution). Otherwise (i.e. Ri is being modified) it calls block BSYAi. Similarly RBi calls MVBi or BSYBi. An A as the penultimate character in a block identifier indicates that the block describes one of the processes that handles register or tag data specified by operand field A. Similar comments apply for B as the penultimate character in a block identifier.

5.2.10 The Blocks MVAi and CHKAi

The block MVAi (i = 1,...,4) moves the contents of FU into TRi. This indicates that the contents of Ri are to be sent to FUx ($x = C(FU)_{10}$), and that its new contents will come from FUx. Concurrently with this, MVAi calls CHKAi. This block checks the value x to determine which function unit is to receive the contents of Ri. It then transfers these data with the register-transfer Dx ← R1.

5.2.11 The Blocks BSYA1 and CHKBAi

Control passes to block BSYAi (i = 1,...,4) in the event that Ri is busy (i.e. is waiting on a result from one of the function units). This block moves the contents of tag TRi to the function unit in lieu of the contents of R1. Next it updates TR1 to indicate from which function unit R1 will receive its new value.

Moving the contents of TRi is actually done by CHKBAi. This block first checks the contents of FU to determine which function unit is to receive the contents of TRi.

5.2.12 The Blocks MVB1 and BSYBi

The block MVB1 (i = 1,...,4) is similar to block MVAi. However, it pertains to operand field B rather than A. A register specified by operand field B of an instruction is used only as a source of data. Therefore its associated tag register is not changed. This accounts for the difference between MVBi and MVAi.

The block BSYB1 (i = 1,...,4) is similar to block BSYAi. Its difference is also a result of it pertaining to operand field B.

5.2.13 The Blocks DCD&EX1 and DCD&EX2

The blocks DCD&EX1 and DCD&EX2 describe the processes that control the operation of FU1 and FU2, respectively. DCD&EX1 waits until both the tag registers (TD1 and TS1) are set to 00 before performing the dyadic operation, specified by the code in OC1, on the contents of the input registers (D1 and S1). The result is placed in register F1. This is then broadcast, conditionally, to registers R1, R2, R3 and R4 as well as to the input registers D2 and S2 of FU2. This broadcast is described by block BCAST1 and is called by DCD&EX1 through the mutual exclusion process DBUS (see section 5.2.7). The process described by DCD&EX2 is similar.

5.2.14 The Blocks BCAST1 and BCAST2

The block BCAST1 describes a broadcast-like process which moves the contents of F1 to any of the registers R1, R2, R3, R4, D2 or S2, whose associated tag registers are set to 01. (These tag registers are TR1, TR2, TR3, TR4, TD2 and TS2, respectively.) Upon the completion of the broadcast the flag ACT1 is set to 0 to indicate to the CS that FU1 is free to be reused. The process described by BCAST2 is similar.

5.2.15 The Remaining Blocks

The remaining blocks are DECTRi, RiF1, DEC2TRi, RiF2 ($i = 1,\ldots,4$) and DECTj1, j1F2, DECTj2, j2F1 ($j = D,S$) (see Figure 5.3(e)). These blocks describe processes that perform the register-transfers required by blocks BCAST1 and BCAST2. Appropriate tag registers are decoded to see which register-transfers are to be performed.

For example, if BCAST1 is active, DEC1TR1 is called (among other blocks), and it decodes tag TR1 to determine whether F1 is to be sent to R1. If TR1 = 01 this register-transfer is to be carried out. This is done by the process

described in block R1F1. R1F1 also resets the contents of TR1 to indicate that register R1 is no longer being modified by an instruction execution.

5.3 Comments on the Example Design

This example design illustrates the capability of the CHDL to describe various types of concurrency, as well as mutual exclusion. These are essential ingredients for any formalism that seeks to characterize multiprocessing systems.

Also note, firstly, that the END symbol was omitted in Figure 5.3. Strictly speaking, the blocks should have been arranged linearly with End as the last symbol in the last block. Secondly, that the "lexical" index used as a shorthand could be included in the CHDL's syntax and interpreted in a way analogous to open subroutines or macros found in programming languages. And finally, that the CHDL could be improved by including the facility for declaring data types (busses, registers, subfields of registers, etc.). The last improvement was made by Smith in [Smi 77], which describes a simulator for a subset of the CHDL.

6. THE SCOPE OF THE CHDL

APL, or some variant, has been adopted by many proponents of CHDLs as a formalism to describe the functional aspects of register-transfer logic. (See for example [Fal 64], [Fri 67], [Hil 73], [Aze 75] and [Fra 75].) Taking this consensus as an acknowledgement of APL's capability to characterize adequately the functional aspects of the DSs of digital systems, we shall only examine the scope of the CHDL with respect to the design of CSs.

One very common model for CSs is the flowchart. This is evident from the large number of digital systems that are controlled by microprograms. The Structure Theorem [Mil 72] shows that any flowchart can be represented as an expansion of the following constructs:

1. f then g
2. if p then f else g
3. while p do f

where f and g are flowcharts with one input and one output, and, then, if, else, while, do are logical connectives. If the term "flowchart" is interpreted as "process" (in our sense), the following consequences arise: f and g can be interpreted as processes described by blocks in the CHDL. This follows since such processes are controlled by a single link, and, hence, can be thought of as having one input and one output (the request and acknowledge signals of the link). Furthermore, the above three constructs are then seen to occur in the CHDL: 3 exists explicitly - the WPROC block, 2 exists in a more general form - the DPROC block, and 1 also exists in a more general form - the order information. Hence, the scope of the CHDL encompasses that of the flowchart model.

In addition to this logical sufficiency the CHDL has considerable scope for parallelism, facilitating the design of high performance systems. Firstly, simple unrestricted parallelism can be described. This qualification refers to the fact that the order information allows parallelism to be described without unnecessarily binding processes together, as is often the case with restricted methods of representing parallelism, such as the use of next or and operators [Bel 71] [Wir 66]. (These two particular operators restrict a language to series/parallel structured processes.) Secondly, overlap or assembly-line type of parallelism can be described using TPROC blocks. Finally, mutual exclusion can be described using MPROC blocks. This allows resolution of some simple resource conflicts that arise as a result of parallelism. (Other resource conflicts, such as shared blocks and shared register-transfers, are handled by SR modules.)

The above discussion suggests that the first of the two purposes of this thesis stated in the Introduction (to develop a CHDL with sufficient scope to describe multiprocessing systems) has been satisfied. Nevertheless, it should be pointed out that more comprehensive models exist. Typical of these is the PN which can describe CSs that are outside the scope of the CHDL. However, much of the additional scope these afford is of questionable use, and it is our opinion that the considerable complexity of any PN that models an entire CS can confuse rather than aid the design process. (Bear in mind that our use of PNs is to define behaviors, and then later to prove assertions about those behaviors. We do not use them as a design aid.)

7. PROOF THAT SYNTACTICALLY CORRECT CHDL PROGRAMS DESCRIBE SYSTEMS WHICH HAVE DEADLOCK-FREE CSS

This chapter introduces some additional syntactic requirements. Then it is proved, using a method for characterizing the behavior of networks of CHDL blocks, that syntactically correct CHDL programs describe systems which have deadlock-free CSs. Computational complexity arguments show that checking the syntax (excluding the APL expressions of the register-transfers) is very simple. It is concluded that the second purpose of this thesis has been met (to specify the CHDL so that syntactically correct programs describe systems which have deadlock-free CSs), without resorting to a complex syntax.

7.1 The Additional Syntax

There are some additional syntactic requirements, not easily expressed by a context-free grammar, that CHDL programs must satisfy. Consequently they were not represented in the syntax of Figure 3.1 but are instead listed below. In the remainder of this discussion the phrase "syntactically correct" (SC) should be interpreted to mean "satisfying the syntax of Figure 3.1 and the additional syntax (AS) below".

AS1. The inter-block connections invoked by the process-call statements must form a partial ordering.

AS2. Every block ID in a process-call statement must have a unique corresponding block.

AS3. The adjacency structure of every PROC and WPROC block must form a partial ordering with a universal lower bound.

AS4. Each statement label must be unique within its block.

AS5. In every DPROC block the bit string in the BITS field of every statement must have the same length.

AS6. If there are less than 2^{ℓ} (ℓ = number of bits in the BITS field) statements in the DLIST of any DPROC, one of them must have None in the BITS field.

AS7. Every statement label in an MPROC block must occur in at least one of the pairs of the mutual exclusion condition.

AS8. Every number in the pairs of the mutual exclusion condition of an MPROC must occur as a statement label.

Notice that these additional syntactic requirements cover those stipulations about the syntax of the CHDL, noted as footnotes in Chapter 3, that were not covered by the contex-free grammar of Figure 3.1.

7.2 The Proof

To prove that SC CHDL programs describe systems which have deadlock-free CSs, freedom from deadlock is defined in terms of process behavior (i.e. in terms of PNs), and then all SC CHDL programs are shown to satisfy this definition.

In Chapter 4 we saw how to derive a PN that defines the behavior of a process described by a CHDL block. This was derived from the block's underlying network of CS modules by using the PNs of the ten CS modules, Construction 2.1, and the simplifications of Chapter 2. In the proof of this section we shall be concerned with whole CHDL programs, i.e. networks of CHDL blocks. In order to define the behavior of networks of blocks we could also join the PNs of the blocks together using Construction 2.1. However, it is convenient to take a slightly different approach.

Consider a block in a network of blocks. Its behavior in such an environment is defined by simulating its PN, B, according to the procedure (modified from that in Section 2.1) below (see Figure 7.1):

P1

1. $\theta \leftarrow T$

2. Choose π, a non-empty subset of π_I.

3. $M(p) \leftarrow 1 \ \forall p \in \pi$

4. Compute the set of enabled transitions in θ (U).

5. Choose one transition t, $\in$ U.

6. If $t \in \tau$ then $\theta \leftarrow \theta - \{t\}$.

7. Fire t.

8. If M(p) = 1 for any $p \in \pi_I - \pi$ then halt.

9. If $\theta \cap \tau = \phi$ then $M(p_{i_1}) \leftarrow \ldots \leftarrow M(p_{i_m}) \leftarrow 0$ go to 1.

10. Go to 4.

where;

$T = \{t_1, \ldots, t_n\}$ the set of transitions in B.

$\pi_I = \{p_1, \ldots, p_k\}$ the set of input places.

$\pi = \{p_{i_1}, \ldots, p_{i_m}\}$ a non-empty subset of π_I

$\tau = \{t_{i_1}, \ldots, t_{i_m}\}$ those transitions that are output transitions of the places in π. See Figure 7.1.

To understand P1 two definitions are necessary. Firstly, the input places* of a block's PN are those corresponding to the controlling processes, or input links. Secondly, the output places* are those corresponding to process-call statements, or output links. The simulation defined by P1 can then be thought of in the following way: A non-empty subset, π, of the input places of B is marked with a token (this represents the action of the environment on the block), the remaining input places are assumed to be empty. B is simulated until all the places in π empty and then refill with a token. These tokens are then removed. A new π is selected at random

*These terms are not to be confused with input and output places of transitions. (See Section 2.1.) It is required, if p is an input/output place with x and y as its input and output transition, respectively, that:
1. x is the only input transition of p.
2. y is the only output transition of p.
3. p is the only input place of y.
4. p is the only output place of x.

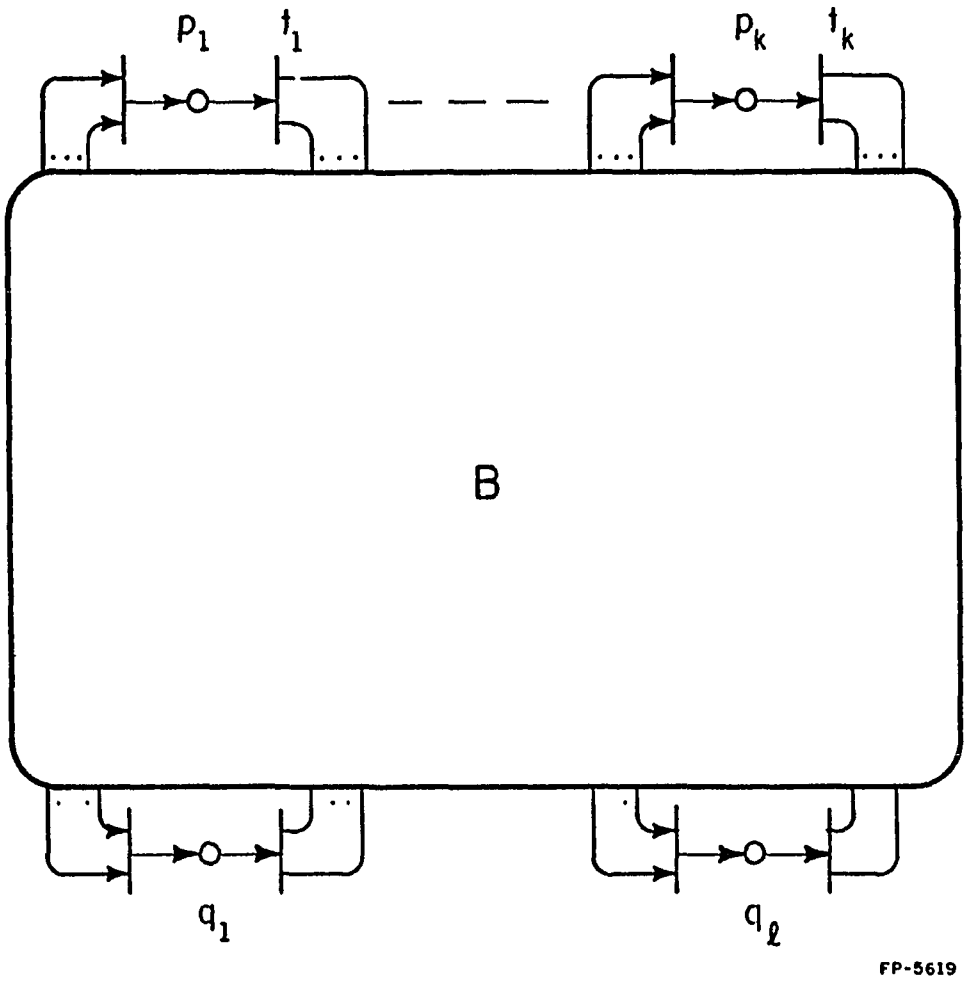


Figure 7.1. A CHDL Block's PN.

(it may be the same as before), these places are marked and the procedure repeats.

The random markings of the input places model the transfer of control from the block's environment to the block. When the block's PN has been simulated and those input places refill, removal of their tokens corresponds to the return of control to the environment. We assume that the initial marking of B does not place tokens in any of the input or output places. Thus under P1, simulation halts if the condition in statement 8 is true, because we require, in order to keep our PN interpretation consistent with the behavior we are trying to model, that input places can only be refilled if they receive a token in the immediately preceding marking phase of the simulation (statement 3).

The output places of B are shown as $q_1,\ldots,q_\ell$ in Figure 7.1. Tokens in these correspond to processes occurring in B's environment that are controlled by B.

Two comments are pertinent regarding the above PN model of the interaction of a CHDL program's block with its environment. Firstly, the concepts of liveness and safeness, defined in Section 2.1, still apply to a PN simulating under P1. Secondly, the behavior of a block's environment may be such that only some of the π's can occur. To reflect this, the choice in statement 2 of P1 can be limited to a subset of the set of all sets of input places. We shall call this subset the environmental constraint (EC).

Freedom from deadlock can now be defined: A CHDL program describes a system whose CS is deadlock-free, if the PNs for all of the blocks of the program, together with their initial markings, are each live and safe (LS) when simulated under P1 with their respective ECs.

This definition is in accordance with an intuitive idea of freedom from deadlock, because P1 never halts, and no places or transitions ever become excluded from the action of P1, if the PNs are LS. Hence, the processes defined by such PNs never reach a point from which they cannot proceed.

Theorem 1: For every SC PROC block $\exists$ a PN* with a single input place, x, an initial marking, $M_o = (0,\ldots,0)$, and an $EC = \{\{x\}\}$, that is LS under P1.

Proof: For every SC PROC block $\exists$ a PN that is a strongly connected marked graph (in a marked graph every place has exactly one input transition and one output transition) that defines its behavior (see Section 4.1). A member of this class of PNs is LS if every circuit in its PN graph has exactly one place containing a token (see [Com 71] for more on marcked graphs).

Since $EC = \{\{x\}\}$, the marked graph that defines the behavior of any PROC block receives a token in x each time P1 executes its statement 2 (the choice for π is limited to $\{x\}$). The input place, x, is in every circuit of the marked graph; therefore, the sufficient condition for LS, mentioned above, is satisfied.

Theorem 2: For every SC DPROC block $\exists$ a PN with a single input place, x, an initial marking, $M_o = (0,\ldots,0)$, and $EC = \{\{x\}\}$, that is LS under P1.

Proof: For every SC DPROC $\exists$ a PN that is a strongly connected state machine graph(in a state machine graph every transition has exactly one input place

*It was noted in section 2.12 that there is, in general, no unique PN graph associated with a particular behavior. Since we are primarily concerned with the behavior, not its defining PN, it is sufficient to consider any one of the set of PNs associated with that behavior.

and one output place) that defines its behavior (see Section 4.2). A member of this class of PNs is LS if exactly one place contains a token (see [Hei 76] for more on state machine graphs).

Since EC = {{x}}, the state machine graph that defines the behavior of any DPROC block receives a token in x each time P1 executes its statement 2 (the choice for π is limited to {x}). Hence, the condition for LS, mentioned above, is satisfied.

Theorem 3: The PN for an ME module with two input places, x and y, an initial marking, $M_o = (0,\ldots,1,\ldots,0)$ (the one initial token is in place S of the ME module's PN - see Figure 2.10), and an EC = {{x}, {y}} or {{x,y}} or {{x}, {y}, {x,y}}, is LS under P1.

Proof: Obvious from Figure 2.10.

Theorem 4: The PN for an SR module with two input places, x and y, an initial marking, $M_o = (o,\ldots,1,\ldots,0)$ (the one initial token is in place S of the SR module's PN - see Figure 2.9), and an EC = {{x}, {y}} or {{x,y}} or {{x}, {y}, {x,y}}, is LS under P1.

Proof: Obvious from Figure 2.9.

Theorem 5: The PN for a T module with a single input place, x, an initial $M_o = (0,\ldots,1,\ldots,0)$ (the one initial token is in place S of the T module's PN - see Figure 2.7), and an EC = {{x}}, is LS under P1.

Proof: Obvious from Figure 2.7.

Theorem 6: The PN for an I module with a single input place, x, an initial marking, $M_o = (0,\ldots,0)$, and an $EC = \{\{x\}\}$, is LS under P1.

Proof: Obvious from Figure 2.12.

Theorem 7: SC CHDL programs describe systems which have deadlock-free CSs.

Proof: SC CHDL programs are acyclic networks of some combination of PROC blocks, DPROC blocks, MPROC blocks, TPROC blocks, and WPROC blocks. These can be viewed, for the purpose of this proof, as networks made up from PROC blocks, DPROC blocks, blocks containing single ME modules (MPROC blocks can be thought of as networks of blocks containing single ME modules- the translation procedure of Section 4.3 ensures that these sub-networks are acyclic), blocks containing single SR modules (the trees of SR modules used when blocks are shared can also be thought of as networks of blocks containing single modules), TPROC blocks (these contain a single T module), and blocks containing single I modules (WPROCs can be thought of as two blocks networks: a block with a single I module followed by a PROC block). The highest level blocks in such networks are blocks containing single So modules (see Section 4.6).

Theorems 1 through 6 demonstrate that the behavior of networks made up from PROC blocks, DPROC blocks, and blocks containing single ME, SR, T, or I modules satisfies our definition of deadlock-free (assuming they are started in the correct initial state), if the respective ECs (expressed in the statement of each theorem) of the PNs defining the behaviors of the blocks are also satisfied.

The ECs of the blocks in Theorems 1 through 6 are satisfied if such blocks are called by blocks whose associated PNs are LS. This condition ensures that the output places associated with the calling blocks behave in a way that is consistent with the apparent environment that P1 creates for a block simulating under it.

The highest level blocks contain single So modules. Their PNs (one is shown together with the appropriate initial marking in Figure 2.3) are LS*. Therefore, since any network described by an SC CHDL is acyclic, it follows in a finite number of steps that all the ECs of the blocks of such networks are satisfied. Hence, our definition of deadlock-free is satisfied, and SC CHDL programs describe systems which have deadlock-free CSs.

Note that throughout the discussion on deadlock it was implicitly assumed that the register-transfer processes were never sources of deadlock, i.e. they took a finite, if unbounded, time to complete. In the next section we shall discuss the "cost", in complexity terms, of specifying the CHDL so that SC programs in it have deadlock-free CSs.

7.3 The Complexity of Checking the Syntax of a CHDL Program

Computational complexity analyses are concerned with the "amount of work" done by algorithms. For the purpose of this and remaining discussions, this is measured in terms of the number of operations which must be performed.

*The PNs for So modules have no input places so it is sufficient to consider them as simulating under the original procedure of Section 2.1.

7.3.1 Checking a CHDL Program Against the Syntax of Chapter 3

The first step in this check of a CHDL program is to take the string of characters representing the design and to partition it into a sequence of tokens, where a token is a string of characters that forms a single logical unit.

The syntax given by the productions of Figure 3.1 can be simplified if the following logical units are tokenized: IDs, LABELs, DREGs, and BITS. The resulting simplified grammar is shown in Table 7.1. Notice that the tokenized entities are now represented by a single generic terminal symbol.

Strings of symbols produced by this grammar can be checked for correctness by the finite automaton whose control state diagram is shown in Figure 7.2. (This implies the tokenized CHDL is a regular language, although it is not characterized by a regular grammar, as can be seen from the simplified grammar of Table 7.1.) The control states are shown as circles, with the start and finish states labelled S and F respectively. A string is accepted as correct if starting in state S there exists a path to F such that the arc labels taken in the order in which they occur in the path agree with the string. Otherwise the string is considered to have a syntax error.

For any input string no input symbol is examined by the above parsing procedure more than once; hence any input string is parsed in a number of operations linearly proportional to the length of the input string. Thus an algorithm for checking any CHDL program for correctness need not have a complexity of greater than $O(n)$, where n is the number of statements in the program.

It might be argued that any such algorithm also has to tokenize the logical units mentioned earlier, and that this could increase the degree

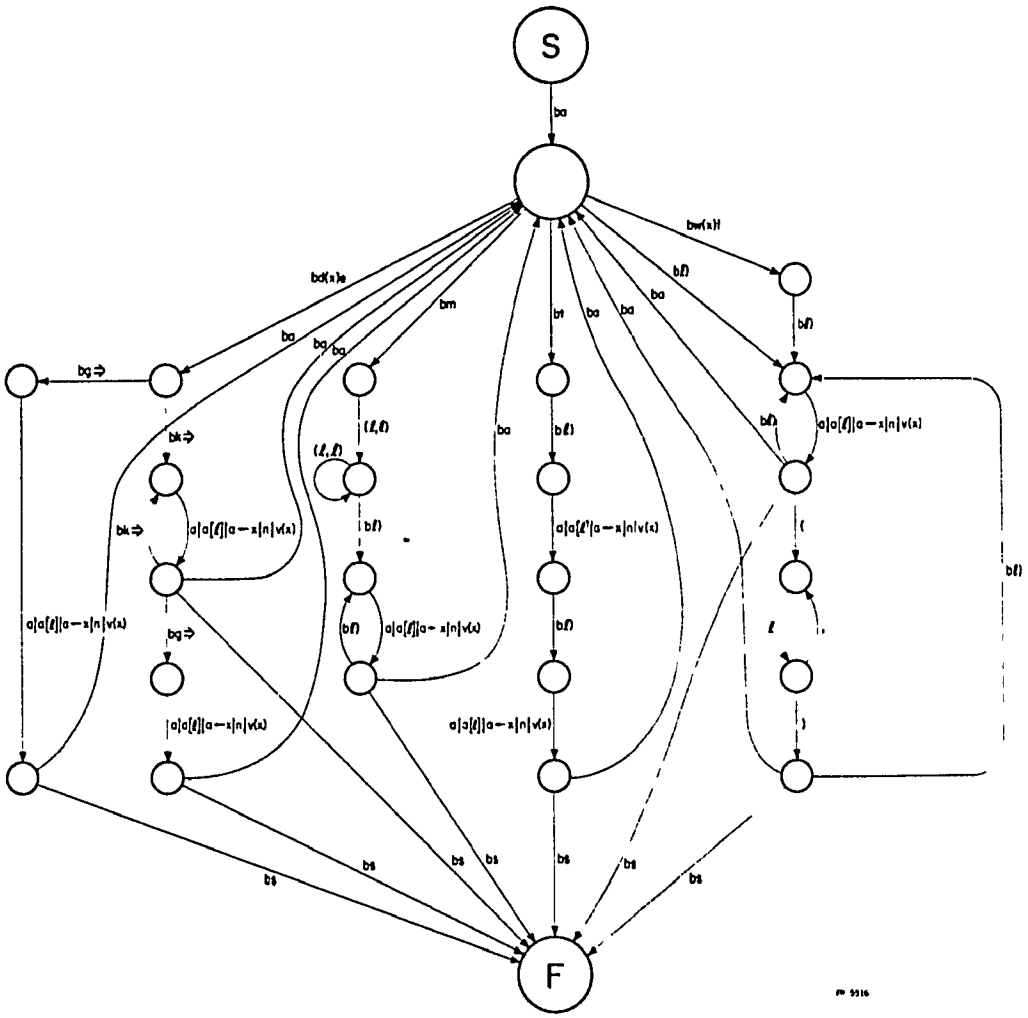


Figure 7.2. The Finite Automaton that Checks the CHDL for Correctness.

```
PGM  ::= { B }b$
  B  ::= baC
  C  ::= P | D | M | T | W
  P  ::= { b1)SI }
  D  ::= bd(x)eA
  M  ::= bm{ (1,1) }{ b1)S }
  T  ::= btb1)Sb1)S
  W  ::= bw(x)fP
  A  ::= bg⇒S | { bk⇒S} | { bk⇒S }bg⇒S
  S  ::= a | a[1] | a←x | n | v(x)
  I  ::= # | (X)
  X  ::= 1 | 1,X
```

Non-Terminals	Equivalent In Figure 3.1
PGM	PROGRAM
B	BLOCK
C	BLOCKBODY
P	PROC
D	DPROC
M	MPROC
T	TPROC
W	WPROC
S	FIELD2
I	FIELD3
A	DLIST
X	ORDER-INFO

Terminals	
b	Dl
$	End
a	tokenized IDs
1	tokenized LABELs
d	Decode
x	tokenized DREGs
e	as
m	Mutex
t	Trigger
w	While
f	do
g	None
k	tokenized BITS
n	Null
v	Wait

Table 7.1. The Simplified Grammar

BLOCK ID	Set of BLOCK IDs that occur as process-call statements in the block at left
B_1	C_1
B_2	C_2
.	.
.	.
.	.
B_u	C_u

Given u blocks with IDs $B_1,\ldots,B_u$

Let $C_i \subseteq B$, where $B = \{B_1,\ldots,B_u\}$

Let $C = \{C_1 \cup \ldots \cup C_u\}$ a multiset (i.e. a set with some repeated elements)

Let $|C| = v$

If $C_i = \{B_{i_1},\ldots,B_{i_k}\}$ the following is true for the inter-block partial ordering:

$$<B_i,B_{i_1}> \ldots <B_i,B_{i_k}> .$$

Table 7.2. Inter-block List.

of complexity of the algorithm. As long as there is a fixed upper bound on the number of characters in a logical unit, this is not the case. Tokenizing a particular string of characters would not be a function of n, but of the fixed bound.

One other point that should be noted is that no account is taken of the fact that DREGs are APL expressions. They are tokenized as single symbols. In other words, our checker does not check their APL syntax. (This would require a separate checker of a more complex type than a finite automaton.) What we are doing is factoring out that aspect of the syntax check associated with the CS and ignoring that associated with the DS. This is compatible with the design approach outlined in the Introduction, where, from the viewpoint of the CHDL, APL expressions which represent functional blocks in the DS are regarded simply as mnemonics for identifying those blocks.

7.3.2 Checking for AS1 and AS2

We require that the inter-block partial ordering invoked by the process-call statements is a true partial ordering (i.e. no circuits are present in the corresponding Hasse diagram, or, to put it another way, no blocks eventually call themselves). Also, we require that every block ID that occurs in a process-call statement has a corresponding block in the CHDL program. This assures us that the processes represented by process-call statements are defined.

The validity of the inter-block partial ordering can be checked using the topological sort algorithm in [Knu 69]. A topological sort takes a partially ordered set (in our case a set of CHDL blocks) and sorts the set into a total ordering such that if $B_i < B_j$ (B_i and B_j are blocks and "<"

is the binary relation "calls" that invokes the partial ordering) is true for the original partial ordering, it remains true for the total ordering. If the sort fails, the set is not partially ordered. In particular its associated Hasse diagram contains at least one circuit. The input to the algorithm is the set of all pairs $<B_i,B_j>$, such that $B_i < B_j$ is true for the partial ordering. In our case this information can be input to the algorithm as a list of the form shown in Table 7.2. Using the notation of Table 7.2, it can be shown that the complexity of the topological sort algorithm (see [Knu 69] for details) is $0(u)+0(v)$. Notice that u equals the number of elements to be sorted (the block IDs, B_i), and v equals the number of pairs $<B_i, B_j>$ defining the partial ordering. This algorithm can be made to abort prematurely if its input is not a true partial ordering. Thus, checking the validity of the inter-block partial ordering can also be achieved by an algorithm of complexity $0(u)+0(v)$, once the input list has been compiled from the CHDL program. The algorithmic complexity of compiling such a list is $0(n)$ (n is again the number of statements in the program): a simple statement-by-statement scan of the program is sufficient. Therefore the overall algorithmic complexity to check for the validity of the inter-block partial ordering is $0(u)+0(v)+0(n)$.

The requirement that every block ID that occurs in a process-call statement must have a corresponding block in the CHDL program can be checked for by an algorithm which confirms that $C \subseteq B$ (see Table 7.2 for notation). This can be done by entering the B_is into a table, then searching for each of the elements in C. The algorithmic complexity of this operation is also $0(u)+0(v)$ (see [Knu 73]).

Both the topological sort and the set inclusion algorithms require suitable hashing methods with the block IDs to achieve the algorithmic complexity measures stated above.

This subsection can be summarized by noting that the algorithmic complexity of checking for AS1 and AS2 of a CHDL program is $O(u) + O(v) + O(n)$.

7.3.3 Checking for AS3 and AS4

We require that the adjacency structure of every PROC and WPROC block be a true partial ordering with a universal lower bound. Also, we require that each number used as a statement label is unique within its block.

These requirements can also be checked using a topological sort on each PROC and WPROC block. The input to the sort is the adjacency structure of the block. This corresponds to the role of the list in Table 7.2: the statement labels correspond to the block IDs, and the order information in FIELD3 of the statements corresponds to the set C_i. Note, however, that these sets are sets of immediate successors in the inter-block partial ordering, whereas the order information represents sets of immediate predecessors in the intra-block partial ordering. This implies only minor changes to the topological sort algorithm referenced in the previous subsection. Checking for the occurrence of at least one empty FIELD3 (this assures us of a universal lower bound; see Chapter 4) and unique labels in every PROC and WPROC block can be done with a simple statement by statement scan. Hence, adopting the arguments of the previous subsection, we get the algorithmic complexity of checking the adjacency structures of a CHDL program as

$$\sum_{i=1}^{\alpha} \{O(p_i) + O(q_i)\} \quad .$$

Here p_i is the number of statements in the i-th PROC or WPROC block, q_i is the total number of labels in the FIELD3s of those statements and α is the number of PROC and WPROC blocks in the program. Notice, as before, that p_i equals the number of elements (the statement labels) to be sorted, and q_i equals the number of ordered pairs of labels defining the intra-block partial ordering.

7.3.4 Checking for AS5 and AS6

We also require that in each DPROC block the bit strings in the BITS fields of every statement are the same length (ℓ), and that, if there are $<2^{\ell}$ statements, one of them begins with None. (We assume the APL expression of the decode argument evaluates to a vector of length ℓ.) These requirements can be checked by a statement-by-statement scan. This leads to a checking algorithm of complexity O(d), where d is the total number of statements in all the DPROC blocks of the program.

7.3.5 Checking for AS7 and AS8

Finally, we require that the mutual exclusion condition in each MPROC block satisfies the following: Every statement label in the block must occur in one of the pairs of the mutual exclusion condition, and every number in a pair of the mutual exclusion condition of an MPROC must occur as a statement label. These requirements can be checked by an algorithm of complexity

$$\sum_{j=1}^{\beta} O(m_j^2),$$

where m_j is the number of statements in the j-th MPROC block, and β the number of MPROC blocks in the program

7.3.6 The Overall Complexity

The results of the five previous subsections can now be combined:

For any CHDL program: $n > u$, $n > v$ and $n > d$.

Also in the worst case for the adjacency structure of a PROC or WPROC block:

$$O(q_i) = O(p_i^2) \quad \text{(See Figure 7.3)}$$

But $n > \sum_{i=1}^{\alpha} p_i$

Therefore $n^2 > \sum_{i=1}^{\alpha} p_i^2$

Similarly $n^2 > \sum_{j=1}^{\beta} m_j^2$

Hence, the overall complexity is given by:

$$O(n^s) \quad 1 < s < 2$$

In most practical cases $s \simeq 1$.

7.4 Concluding Comments

From the result in Section 7.2 it can be seen that the second purpose of this thesis (viz. to specify the CHDL so that SC programs describe systems which have deadlock-free CSs) has been met. From the results of Section 7.3 it can further be seen that it has been met without resorting to a complex syntax for the CHDL or limiting its scope (this last point from Chapter 6).

Freedom from deadlock, coupled with a simple syntax, is achieved by specifying the CHDL so that: different types of processes are separated into different blocks; a process can only be controlled through a single port (except a mutual exclusion process), variously seen as a link, output and input place pair of interacting PNs, or a process-call statement

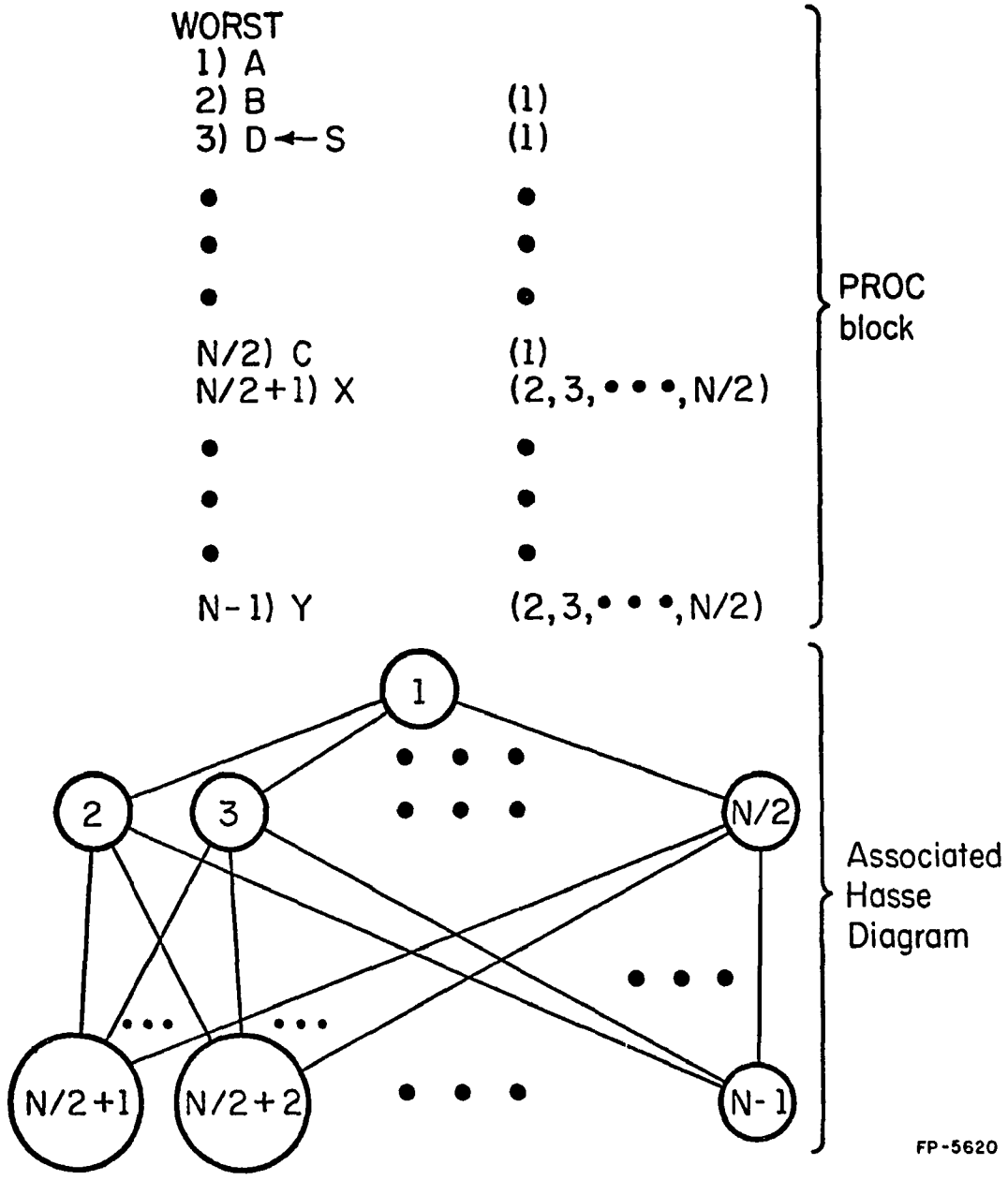


Figure 7.3. A Worst Case Adjacency Structure.

and block ID pair; and the process interaction structure is acyclic. The special case of a mutual exclusion process is specified so that it represents just a small departure from the above scheme. It can be regarded as a set of processes that are each controlled through single ports where the flow of control through neighboring ports in the set can be mutually regulated by the setting and resetting of semaphores.

8. HARDWARE IMPLEMENTATION OF THE CHDL PROGRAMS

Until now little has been said about the logic gate level of implementation of the CS modules, of the functional blocks used in the register-transfers of the DS, and, hence, of the CHDL programs. In this chapter two approaches to the hardware implementation of the CHDL programs are discussed. The first discusses implementing them directly according to the asynchronous model of Chapter 1. The CS modules of Chapter 2 are constructed from logic gates, and the functional blocks of the DS are designed with additional logic to generate acknowledge signals. The second discusses implementing them in a pseudo-asynchronous fashion. This is, strictly speaking, a synchronous realization as a clock is used, but it retains many of the characteristics and advantages of the asynchronous model of Chapter 1.

8.1 Asynchronous Implementation

The most obvious method for implementing a program in the CHDL, at the logic gate level, is to design the CS modules as asyncyronous machines, then interconnect them to form the CS that results from applying the translation procedure of Chapter 4. The APL expressions that define the functional blocks of the DS can be realized as combinational logic with additional logic to generate acknowledge signals.

Designing each of the ten modules need only be done once, but first a signalling convention to define the request (R) and acknowledge (A) signals must be chosen.

There are three seemingly natural conventions (see Figure 8.1).

1) Pulse signalling: R and A can be pulses (see top of figure).
2) Simple signalling: R and A can be transitions from 0 to 1 and 1 to 0 (see center of figure).

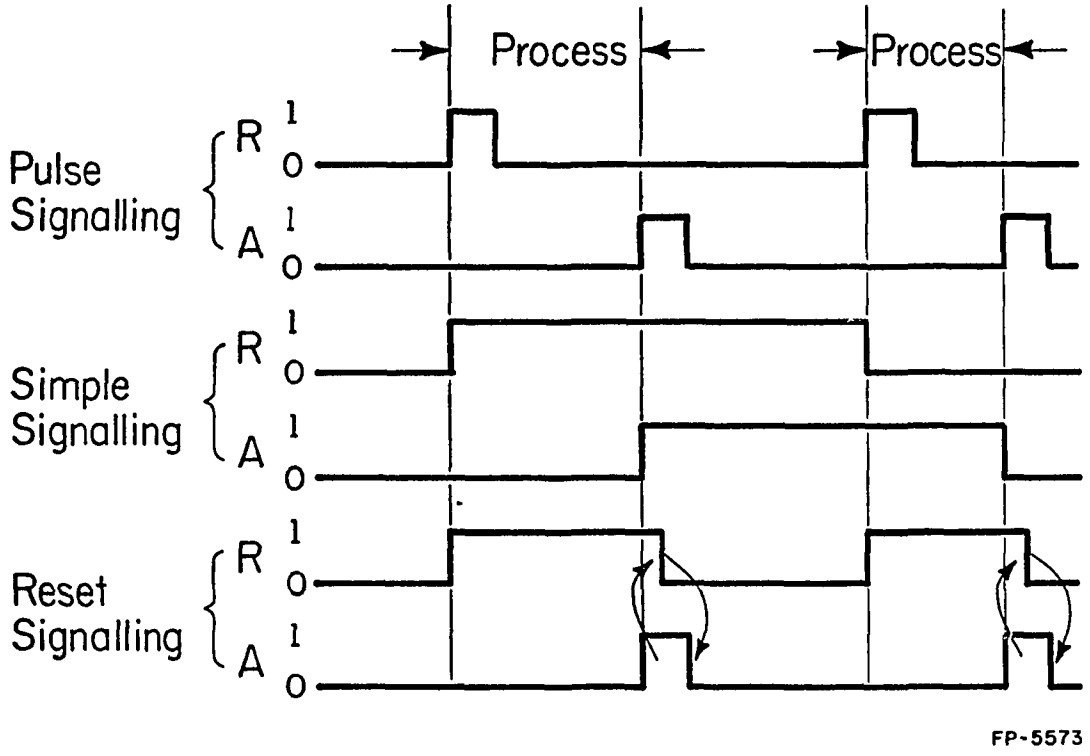


Figure 8.1. Some Signalling Conventions.

3) Reset signalling: R and A can be transitions from 0 to 1 which must be reset to 0 (see bottom of figure).

Most of the modules have already been designed by Peterson using reset signalling [Pet 74] and by Patil using simple signalling [Pat 72]. Figure 8.2 shows eight of the ten modules implemented for simple signalling. The C-element is a one state sequential machine which can be realized by four (two and three input) NAND gates. (See [Mul 63] and [Mil 65] for a further discussion of the C-element and speed independent logic, a logic design methodology that uses simple signalling.) The operation of the modules in Figure 8.2 can be understood from their behaviors in Chapter 2, the simple signalling convention shown in Figure 8.1 and the operation equation of the C-element shown at the bottom of Figure 8.2. (The C-element retains its previous state as long as its two inputs do not agree with each other, but tends towards the state of the inputs whenever they are both the same.) The implementations shown in Figure 8.2 are (except for the I module) all to be found in [Pat 72]. We have included them to give an idea of the complexity of a system's CS at the gate level. The gate level complexity of the modules when implemented for reset signalling is of a similar order [Pet 74]. Nobody has designed any of these CS modules for pulse signalling to our knowledge, although they could be designed using the techniques of [Kel 74]. For various practical reasons pulse signalling is not a very good design choice (in particular, maintenance of pulse integrity makes mono-stables necessary - it has been remarked that the quality of a design is inversely proportional to the number of mono-stables it uses). The design philosophy for implementing modules with simple signalling is discussed in [Den 71], and general design methods for

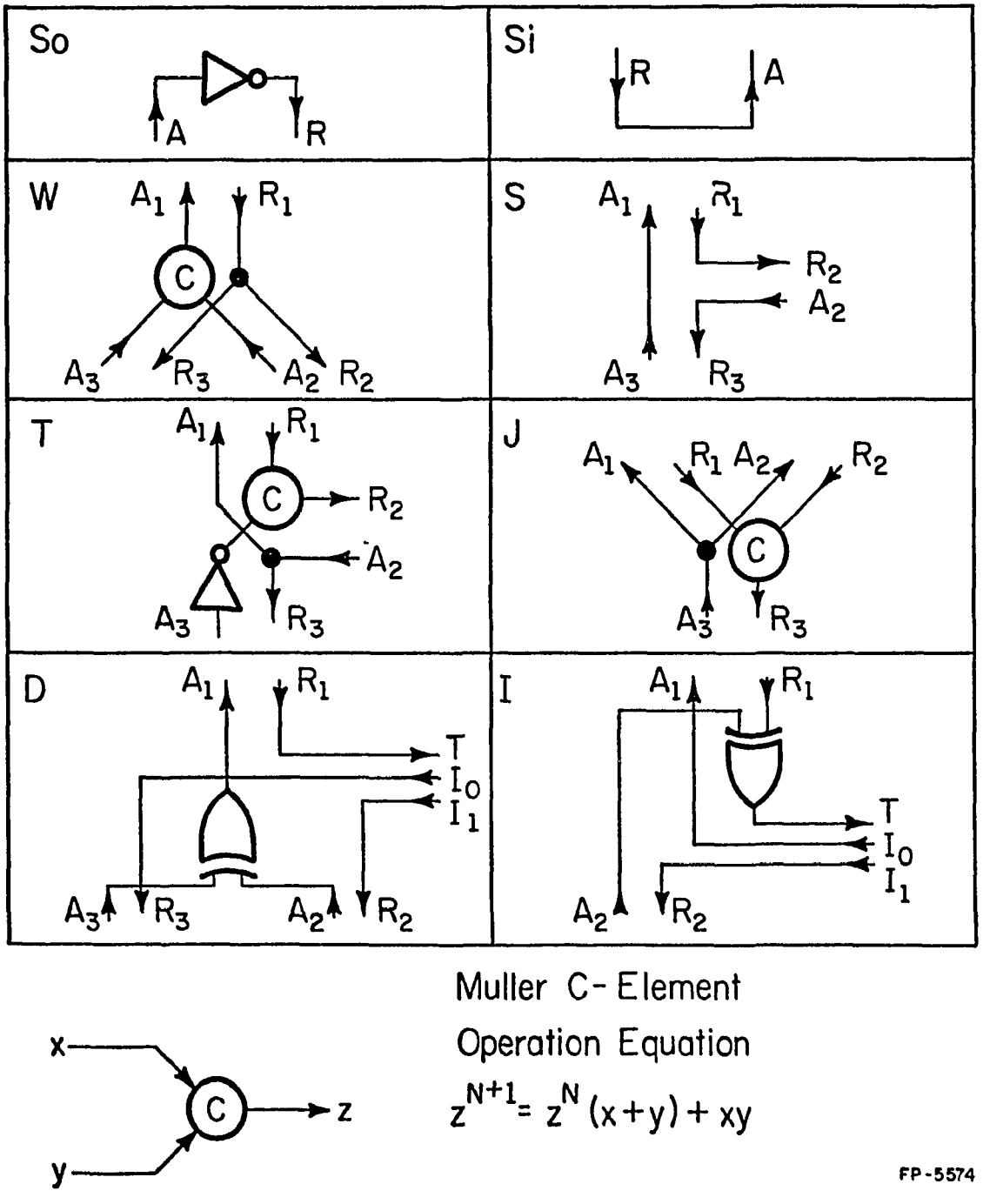


Figure 8.2. Modules using Simple Signalling.

asynchronous modules are discussed in [Alt 69] and [Kel 74], as was also noted in Section 2.13. Keller in [Kel 74] also discusses the problem of multiple signal changes that can occur at the inputs to some of the modules. In our case this point is relevant to the design of the ME and SR modules (not shown in Figure 8.2), where it is possible that both input links have request signals occurring on them simultaneously, each of which requires a different response. This implies a form of arbitration (the J module can experience simultaneous input changes, but no arbitration is needed in its case). Keller presents an arbitration module called the arbitrating test-and-set (ATS) module that can be used to design the ME module. Figure 8.3 shows the ATS module and a state diagram describing its behavior. The ME module can then be implemented for simple signalling as shown in Figure 8.4. The SR module can be implemented directly from the ME as shown in Figure 8.5. Implementing the ATS module is not straightforward, and details can be found in [Kel 74]. In particular, the possibility of multiple input changes (the occurrence of T and R simultaneously in state i) can cause any implementation to get into a metastable state. This phenomenon is further discussed in [Cat 66] and [Cha 73].

Constructing the CS of a system as a network of modules creates a structure which is not readily modified. In many systems the capability to modify the CS, or the more powerful capability of emulation, is required. In such cases the CS can be implemented directly from its PN behavior graph as a programmable logic array [Jum 74] or as a diode array [Pat 75]. The PN graph can be obtained by first applying the translation procedure of Chapter 4 to the CHDL program to get the network of modules that form the CS, then using the Construction 2.1 and the simplifications of Chapter 2 to construct the CS's PN from each module's PN. Unfortunately both the diode

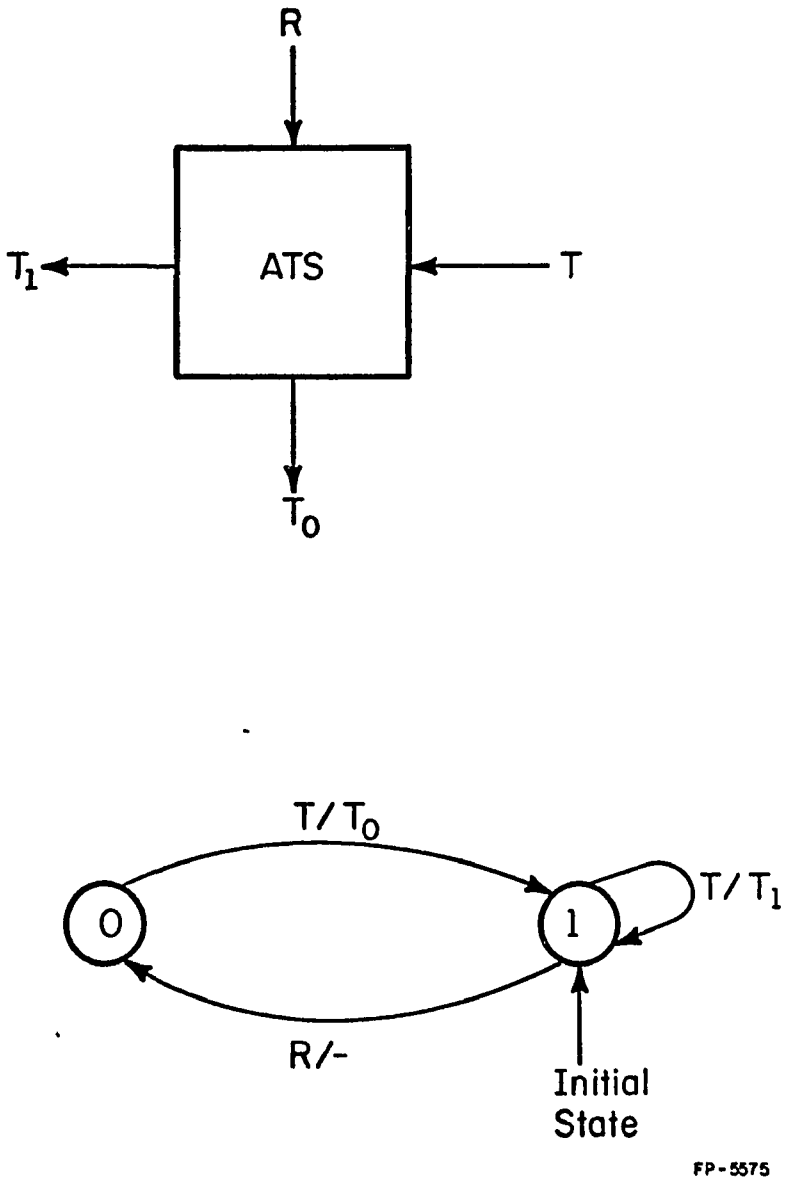


Figure 8.3. The ATS Module.

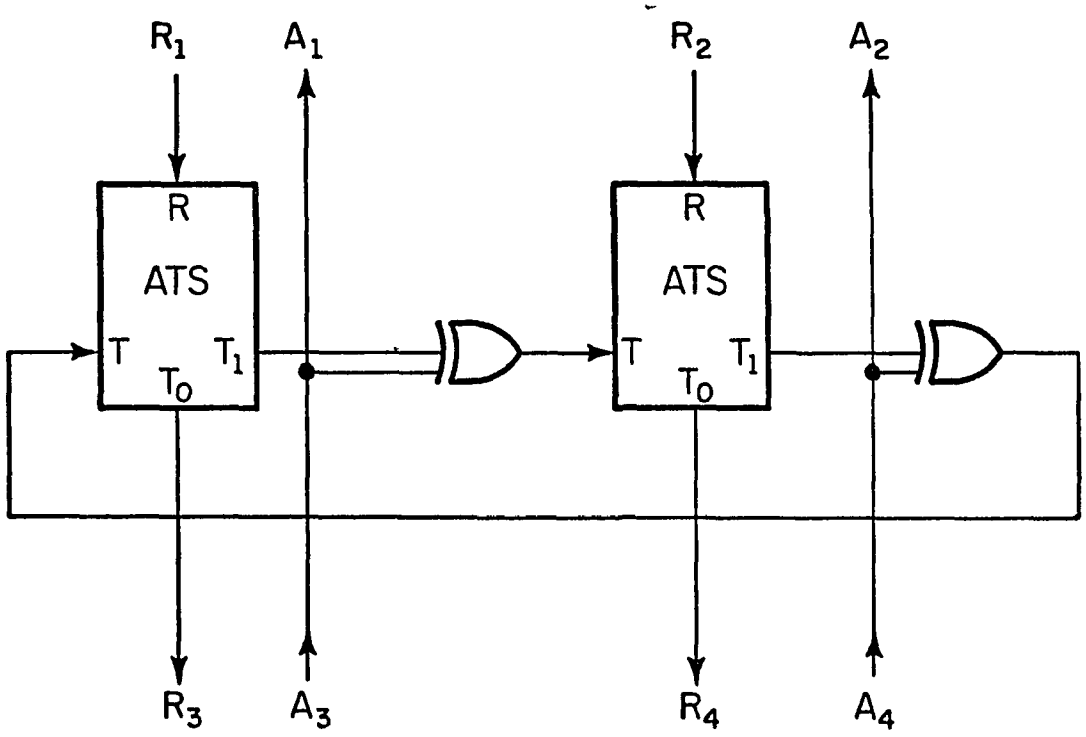


Figure 8.4. The ME Module (Simple Signalling).

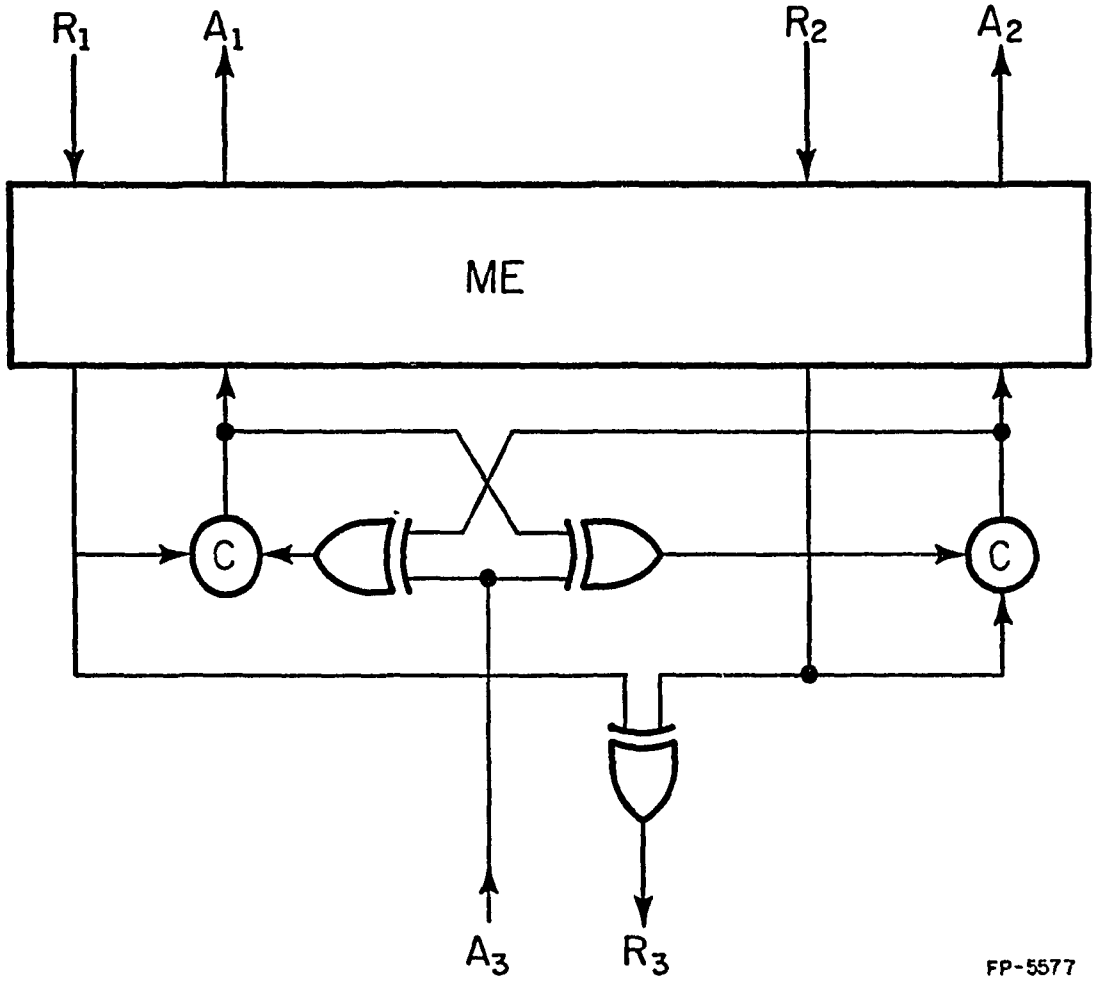


Figure 8.5. The SR Module (Simple Signalling).

array and the programmable array are very inefficient realizations. The diode array uses a flip-flop plus additional logic for each place and each transition in the PN, over and above the diodes. The programmable logic array uses a flip-flop, a C-element plus additional logic in each programmable cell of the array, many of which are programmed just to pass signals between their boundaries without modifying them. Both arrays need external arbiters to implement the equivalent of the SR and ME modules (reducing somewhat their facility for being modified). The programmable array also needs to be able to implement the equivalent of the D and I modules. This can be done with some simple (also programmable) logic on the inputs and outputs of the array.

As noted at the beginning of this section, the functional blocks require additional logic to generate acknowledge signals. To illustrate how this can be done two examples are shown in Figures 8.6 and 8.7. In both reset signalling is assumed. The first shows a simple register-transfer in a bus structured environment, and the second a register-transfer which results in addition or subtraction ($Z \leftarrow X \pm Y$). The operation of the first should be clear from the figure - equivalence gates are used to detect when the content of the destination register is the same as that of the source register. The operation of the second is a little more complicated. It is a modification of a carry completion adder (see [Gsc 75] for details). The outputs of the adder/subtractor that indicate a carry (C) or no carry (N) are also used to generate the acknowledge signal. Correct operation is assured only if A does not occur before the result of the adder/subtractor is latched into the Z register. This need to analyse the timing of the functional blocks to assure correct operation can lead

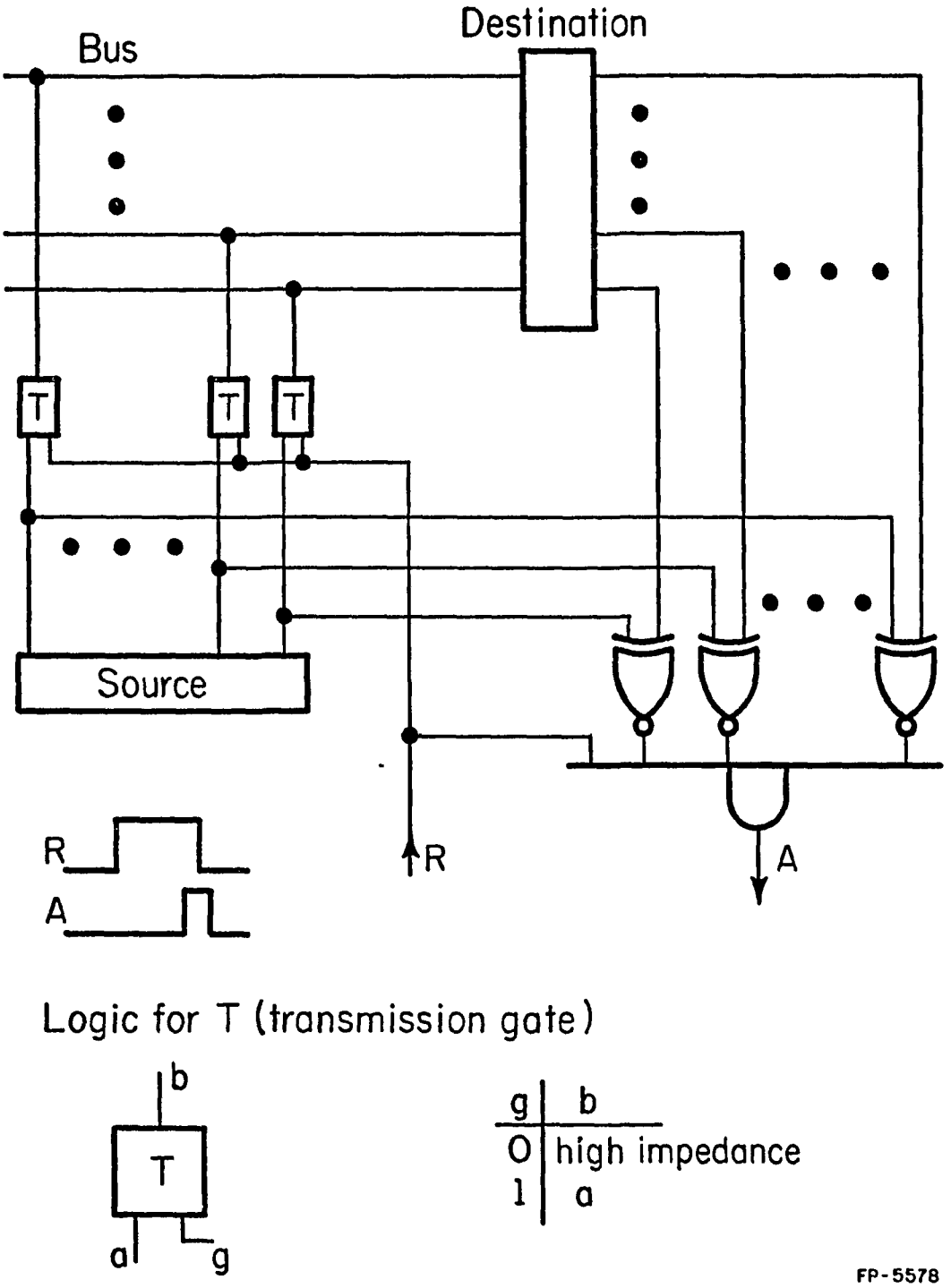


Figure 8.6. Acknowledge Signal Generation 1.

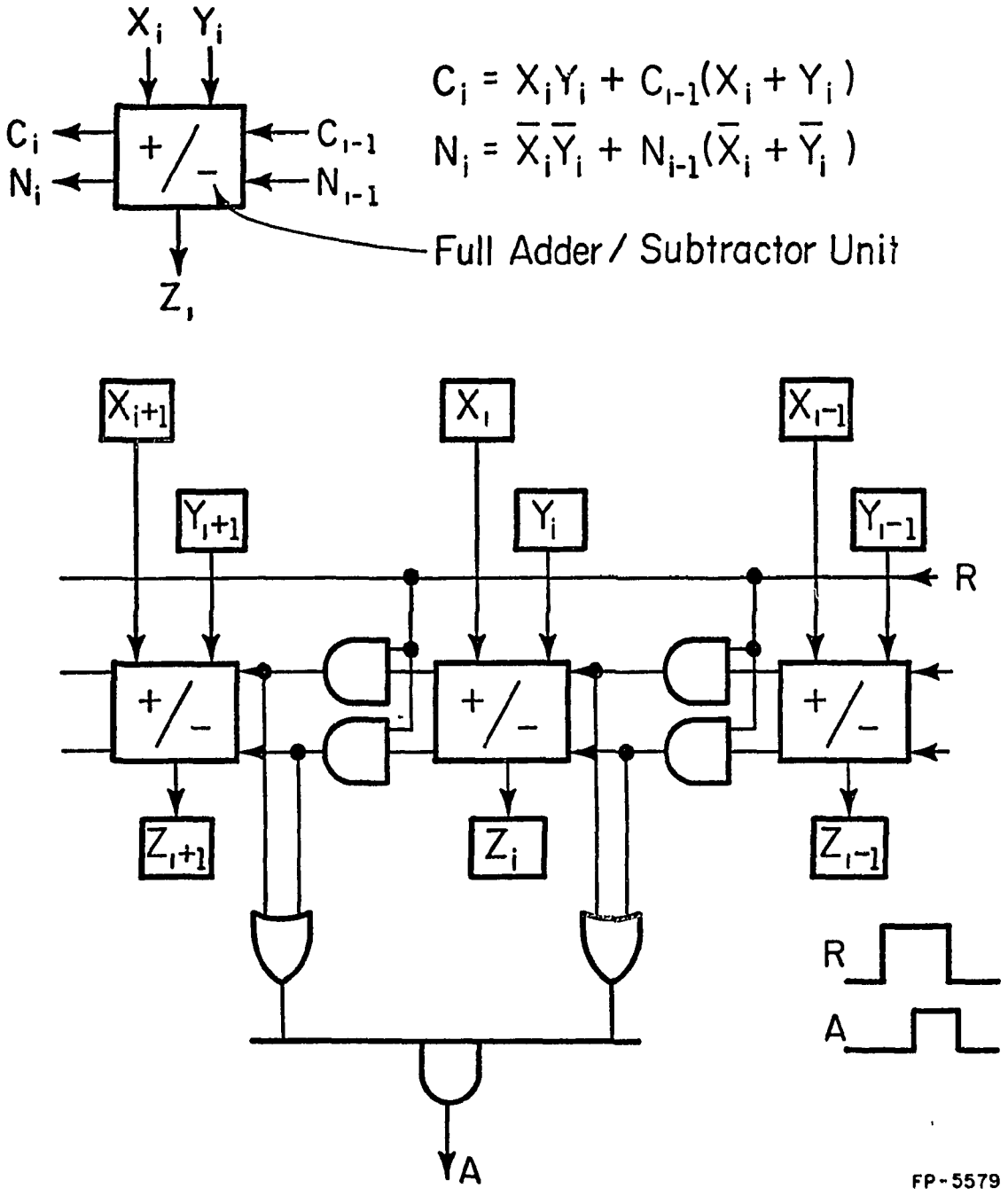


Figure 8.7. Acknowledge Signal Generation 2.

to a complex design procedure (the same considerations apply when designing the CS modules, as is pointed out in [Den 71]; however, the CS modules only have to be designed once whereas each new CHDL program may have many new register-transfers to be designed). The major problem is avoiding delay hazards which can cause premature acknowledge signals to occur. A systematic method of design which results in designs that are free of delay hazards uses a spacer word between each data word.* Unger in [Ung 69] discusses this design method in detail. Although such a systematic approach to DS design is desirable, the loss in throughput rate as a result of including spacer words every other word in the data flow brings into question the speed-up gained at the register-transfer level by operating asynchronously. It should be born in mind that only register-transfers whose time of operation are very data dependent (i.e. not simple "move contents of register A to register B" type register-transfers) result in a faster average time of operation by indicating their own completion rather than having the DS assume a worst case bound.

One final note on the asynchronous implementation concerns fault tolerance. If the CS modules are implemented for simple signalling, any CS constructed from them will automatically halt if a stuck-at fault occurs on the wires connecting the NOTs, EORs, C-elements and ATS modules together. If the CS modules are implemented for a reset signalling using the designs given in [Pet 74], any CS constructed from them will halt if a stuck-at fault occurs on the wires interconnecting the modules. To make the DS fault tolerant many of the usual schemes can be used (see [Sel 68]

*Arrival of a spacer word at the output of a functional block indicates that the combinational logic has been flushed of any delayed logic signals and, hence, is ready to receive new input data.

for examples). However, using m-out-of-n codes offers some interesting bonuses, as the self-checking checkers that can be devised for such codes (see [Smi 77] for more details) can also be used to generate acknowledge signals. If a fault causes a non-codeword, or the checker fails, no acknowledge signal is returned to the CS resulting in its halting.

8.2 Pseudo-asynchronous Implementation

In the previous section we noted some drawbacks associated with asynchronous implementation. These were:

1) The DC can be difficult to design because acknowledge signal generation must be implemented. Furthermore, solutions to this problem do not lend themselves to efficient realization in standard logic families, as these are oriented towards synchronous environments.
2) The CS cannot be implemented efficiently in a way that it can be readily modified.
3) Simultaneous multiple input changes on ME and SR modules can result in non-standard operation of logic elements used in their implementation.

These drawbacks can be overcome by using a central clock to regulate signal changes within a system, while still retaining the essentially asynchronous action described by the CHDL. We use the term pseudo-asynchronous (PA) to describe such implementations.

The system model for PA implementation is shown in Figure 8.8. It is based on one proposed by Glushkov in [Glu 65] and comprises two cooperating finite state machines. One, the CS, is a Mealy machine, and the other, the DS, is a Moore machine. The inputs to the CS are shown as the vector $\underline{X}$, and they represent information about the state of the DS.

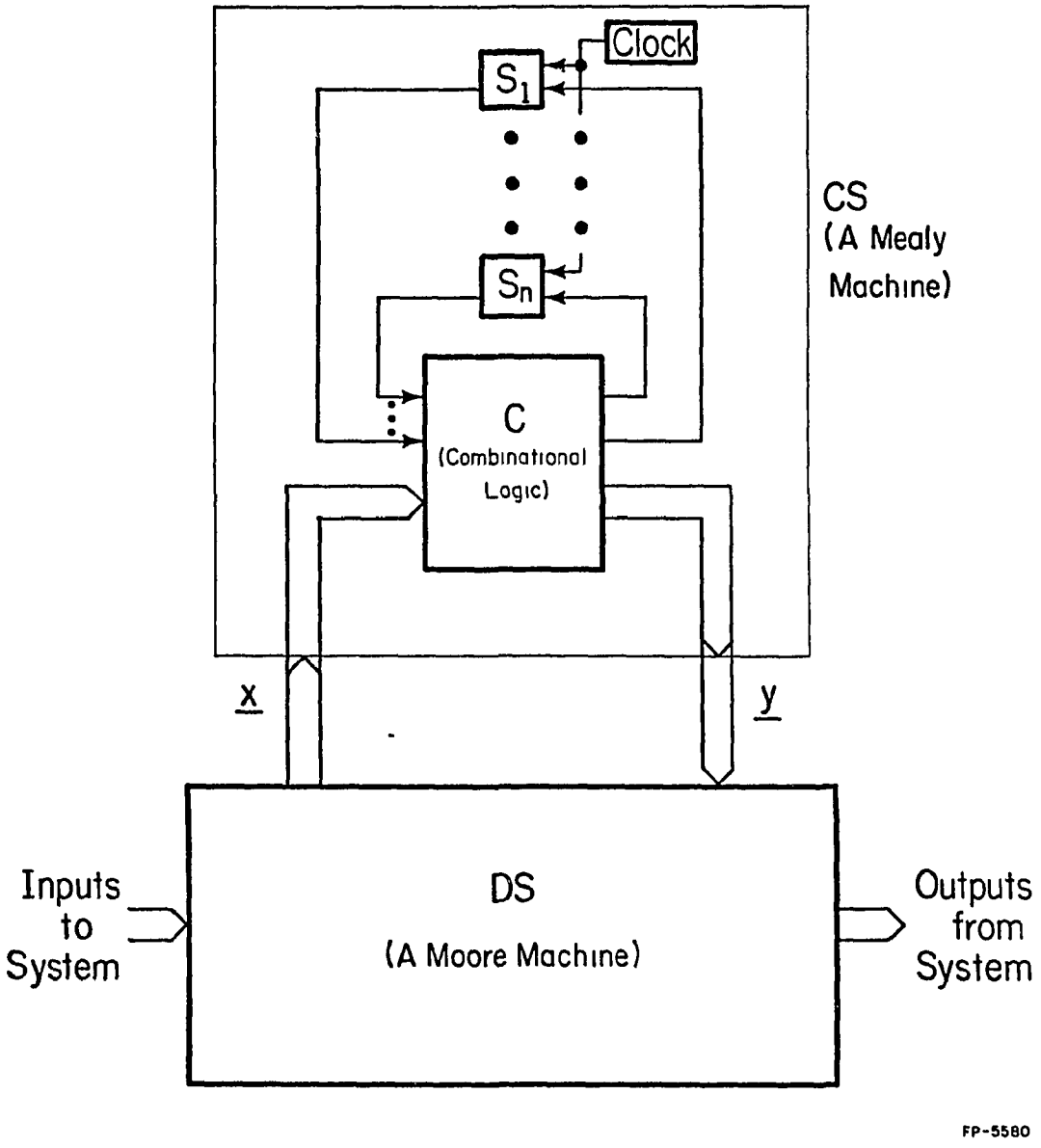


Figure 8.8. The PA System Model.

Based on this information and on its own state (given by s_1 through s_n) the CS machine outputs a set of control signals, shown as the vector $\underline{Y}$. These are gating signals for synchronous register-transfers. No acknowledge signals are generated by the DS logic; instead each register-transfer is allocated a fixed number of basic clock cycles. The number allocated is based on the worst case time for the register-transfer.

Recalling the list of drawbacks associated with the asynchronous implementation, we see that the above PA model overcomes them as follows:

1) The DS no longer needs to include acknowledge signal logic and can be constructed in an efficient way from available logic families with the aid (if necessary) of the many automatic design packages aimed at conventional functional block implementation.
2) The CS can be made easy to modify by realizing C (see Figure 8.8) as a PLA, a ROM, or, if frequent emulation is required, a RAM.
3) The multiple input change problem, that can result in non-standard operation of logic elements does not occur in a synchronous environment. However, the problem still occurs at the interface between the system and its environment, since signals that meet at this boundary are asynchronous with respect to one another.

Nevertheless, using a PA implementation has drawbacks of its own. These are as follows:

1) Operations at the register-transfer level take a fixed worst case time period.

2) The fault tolerance of the asynchronous implementation to many stuck-at faults is lost.

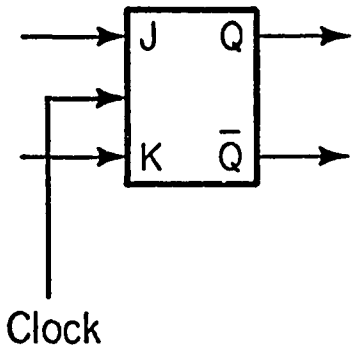
3) Consideration must be given to the layout of the logic gates, so that clock skewing, due to line delays, does not occur. Layout (in particular maximum line length) also limits the speed of the clock and hence of the system.

Without going into a general translation technique we shall present some examples of how the Mealy machine that implements the CS of a system described by a CHDL program can be derived from that program.

The states of the CS(s_1 through s_n) are held in a set of master-slave JK flip-flops (JKFF). Figure 8.9 shows the PA signalling convention. In the case of a register-transfer there is no acknowledge - a counter is used to measure the time out for the register-transfer, and completion of the count serves in lieu of an acknowledge. (The truth table of the JKFF is included in Figure 8.9 for convenience.)

A logic circuit useful in understanding the examples that follow is the "sequential AND" ($). Its operation is shown in Figure 8.10. Its diagrammatic representation is shown at the top, its logic realization is shown in the center, and a timing diagram showing its operation under the three possible sets of inputs is shown at the bottom. It outputs a signal on z after one has occurred on both x and y. (We assume that any two consecutive signals on x (y) are separated by one on Y(9).)

Figure 8.11 shows the PA implementation of the CS of a PROC block. Each statement is associated with at least one JKFF. The JKFFs are the boxes labelled P, 1, 2, 3.1, 3.2 and 4. For clarity the clock lines are omitted, the input at the top of each box is assumed to be J, and the



J	K	Q^{n+1}
0	0	Q^n
0	1	0
1	0	1
1	1	$\overline{Q^n}$

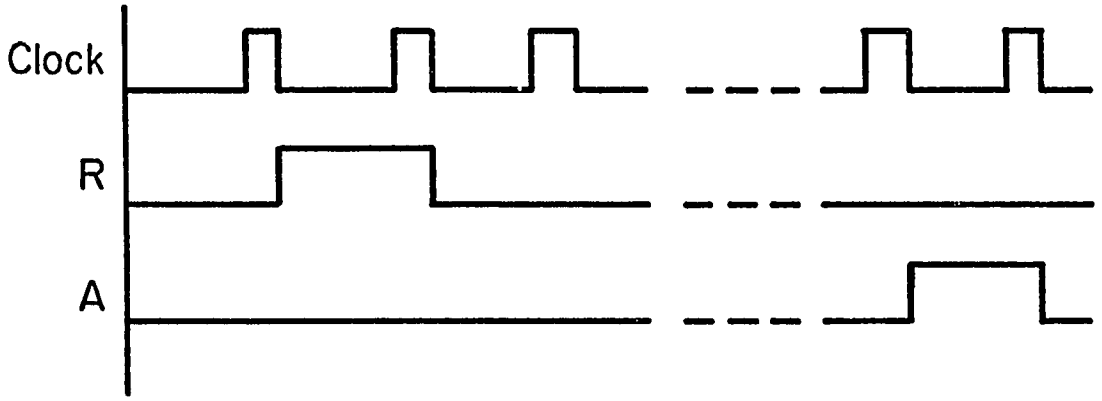


Figure 8.9. PA Signalling Conventions.

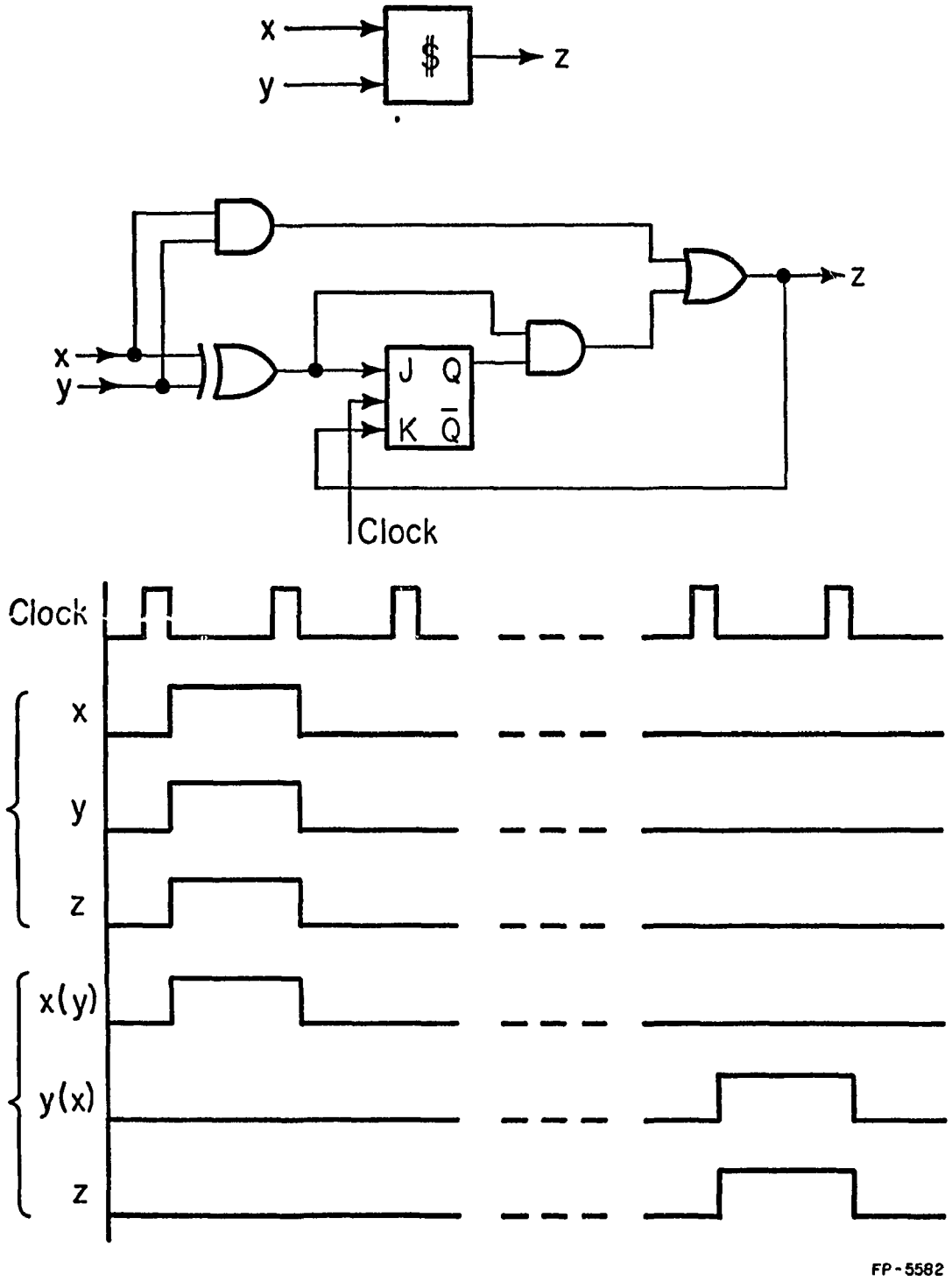


Figure 8.10. The Sequential And.

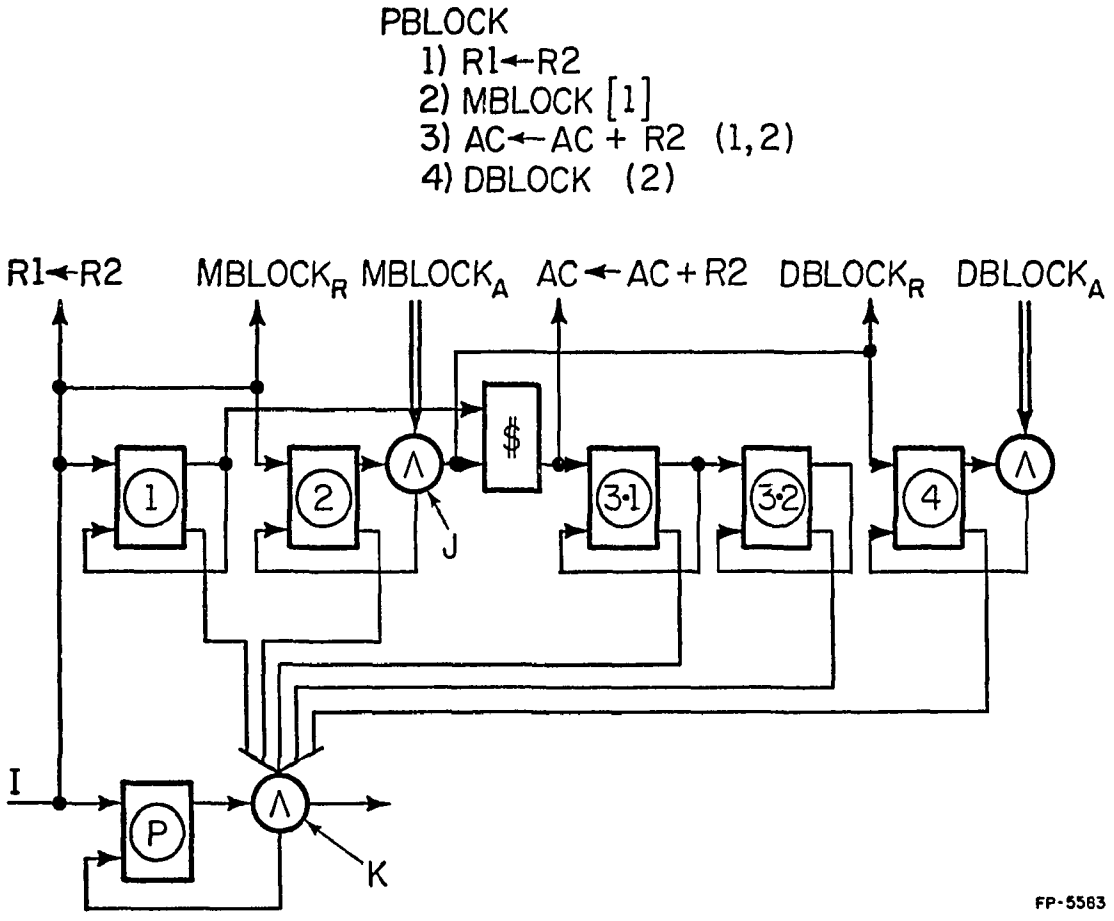


Figure 8.11. PA Implementation of a PROC Block.

output at the top of each box is assumed to be Q. A signal (see R, Figure 8.9) at I starts the machine we have called PBLOCK by gating a register-transfer, R1 ← R2, initiating another machine called MBLOCK with signal $MBLOCK_R$, and setting the JKFFs P, 1 and 2. A set JKFF P indicates that machine PBLOCK is active, a set JKFF 1 indicates that the register-transfer R1 ← R2 is active, and a set JKFF 2 indicates that machine MBLOCK is active. JKFF 1 is reset after one clock period: this is the time for operation allocated to the register-transfer. JKFF 2 is reset after the inputs to the AND gate J go to logic 1. These are labelled $MBLOCK_A$ and are the $\overline{Q}$ outputs from the JKFFs that would be used to implement the CS of MBLOCK. (They stand in the same relation to 2 as the inputs to AND gate K stand to P.) The signal $MBLOCK_R$ corresponds to R of Figure 8.9, and the outputs of AND gate J correspond to A of Figure 8.9. The register-transfer AC ← AC + 1 receives a gating signal from the output of the $ gate as soon as either the MBLOCK machine is done, or the time allocated the register-transfer R1 ← R2 is up, whichever takes longest. The register-transfer is allocated two clock periods to complete, which are counted by JKFFs 3.1 and 3.2. The signal from J also starts machine DBLOCK, setting JKFF as it does so. When DBLOCK is done JKFF 4 is reset in a similar fashion to JKFF 2. When PBLOCK is done the JKFFs 1,2,3,1, 3.2 and 4 are reset. This enables AND gate K which causes P to be reset after the next clock pulse. A reset JKFF P indicates that PBLOCK is done.

Figure 8.12 shows the PA implementation of the CS of a DPROC block. Its operation should be clear from the previous discussion. The combinational logic with inputs x_1 and x_o directs the start signal to the appropriate

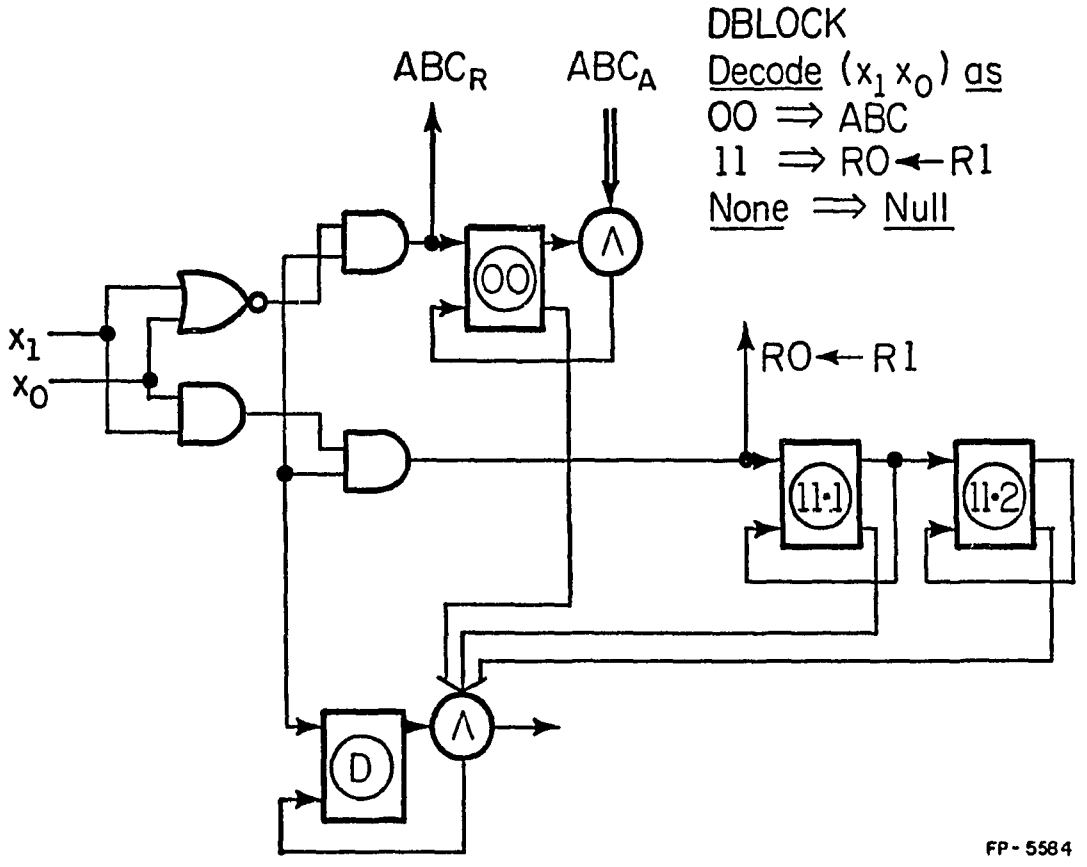


Figure 8.12. PA Implementation of a DPROC Block.

submachine. Notice that the register-transfer R0←R1 has been allocated two clock periods. In the case of $x_1 \oplus x_0 = 1$ JKFF D is set and then reset after the following clock pulse - a Null process.

Figure 8.13 shows the PA implementation of the CS of a WPROC block. Again, its operation should be clear from the previous discussion. The output signal from AND gate L is used to reinitiate the machine if $y = 1$ is true.

Figure 8.15 shows the PA implementation of the CS of a TPROC block. The inclusion of the $ gate with its JKFF initially set (see Figure 8.10) allows machine XYZ to be active while the process of which TBLOCK (represented by JKFF T in the implementation) is a part, may be reinitiated. This reinitiation may proceed until just before TBLOCK. It must then wait until XYZ is done. Thus the overlap never goes beyond one level.

Figure 8.14 shows the PA implementation of the CS of an MPROC block. We have arbitrarily given priority to process TWO. This is determined by gate M.

In a complete CS, many of the JKFFs are redundant. The only essential ones are those associated with $ gates and the register-transfer timing. The redundant ones can be eliminated or retained for use in system diagnosis. The JKFFs which form the state vector, s_1 through s_n, may be regarded as a control status word (CSW) which must be initialized to start the machine (usually most of the JKFFs are reset, but those in $ gates associated with TPROCs are set). This CSW may also be set to intermediate values as part of a diagnostic routine.

It can be seen from this brief sketch of PA implementation that many of the characteristics of the asynchronous model are retained,

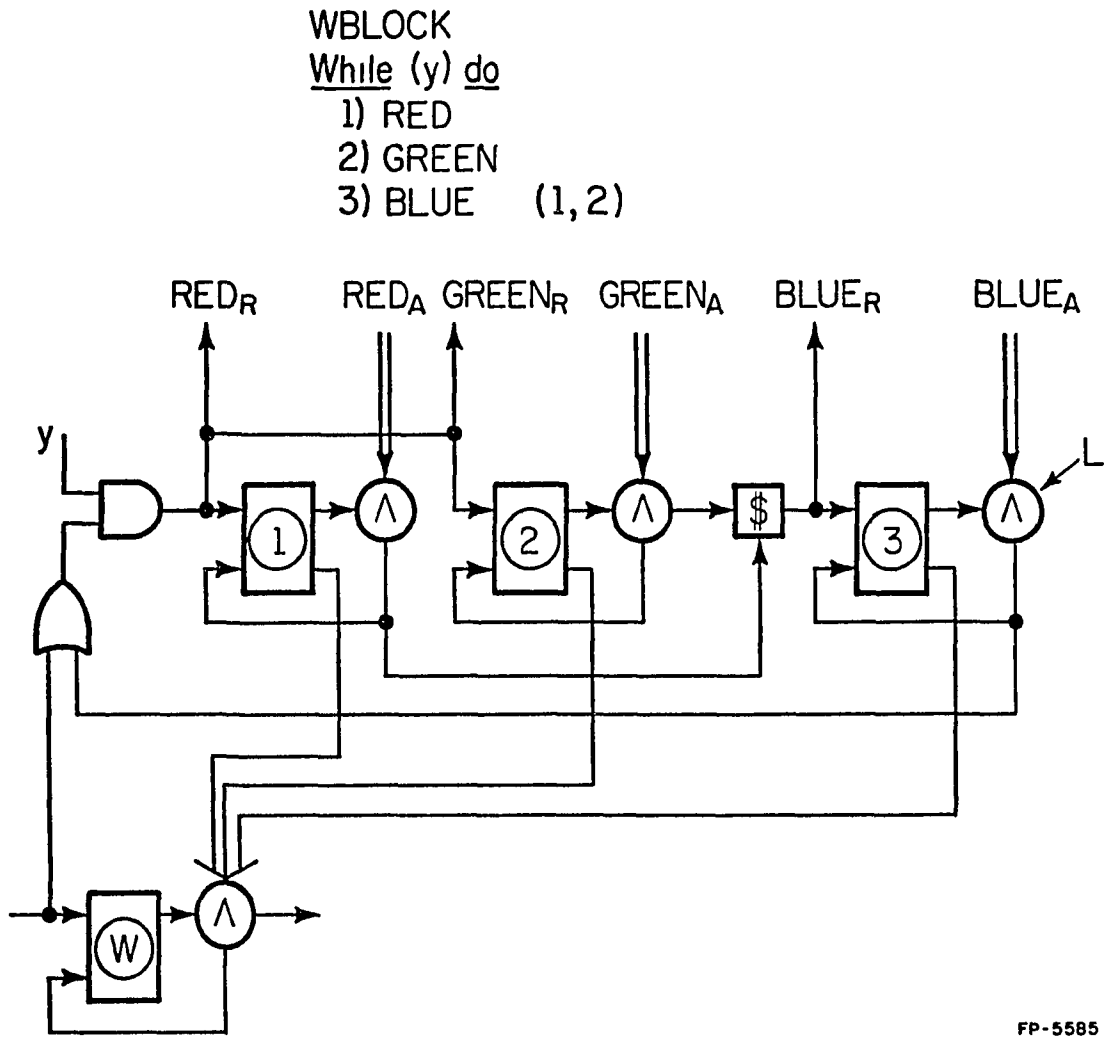


Figure 8.13. PA Implementation of a WPROC Block.

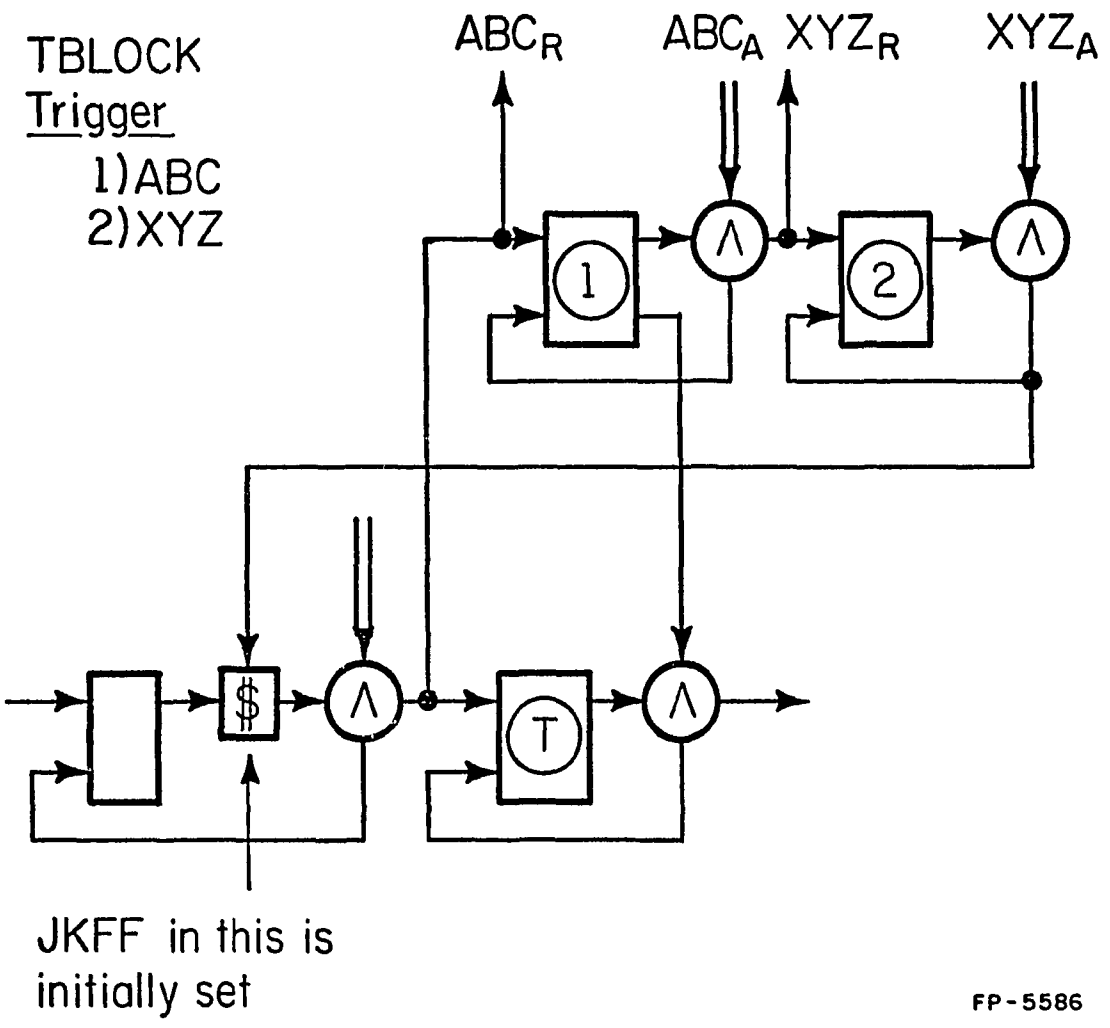


Figure 8.14. PA Implementation of a TPROC Block.

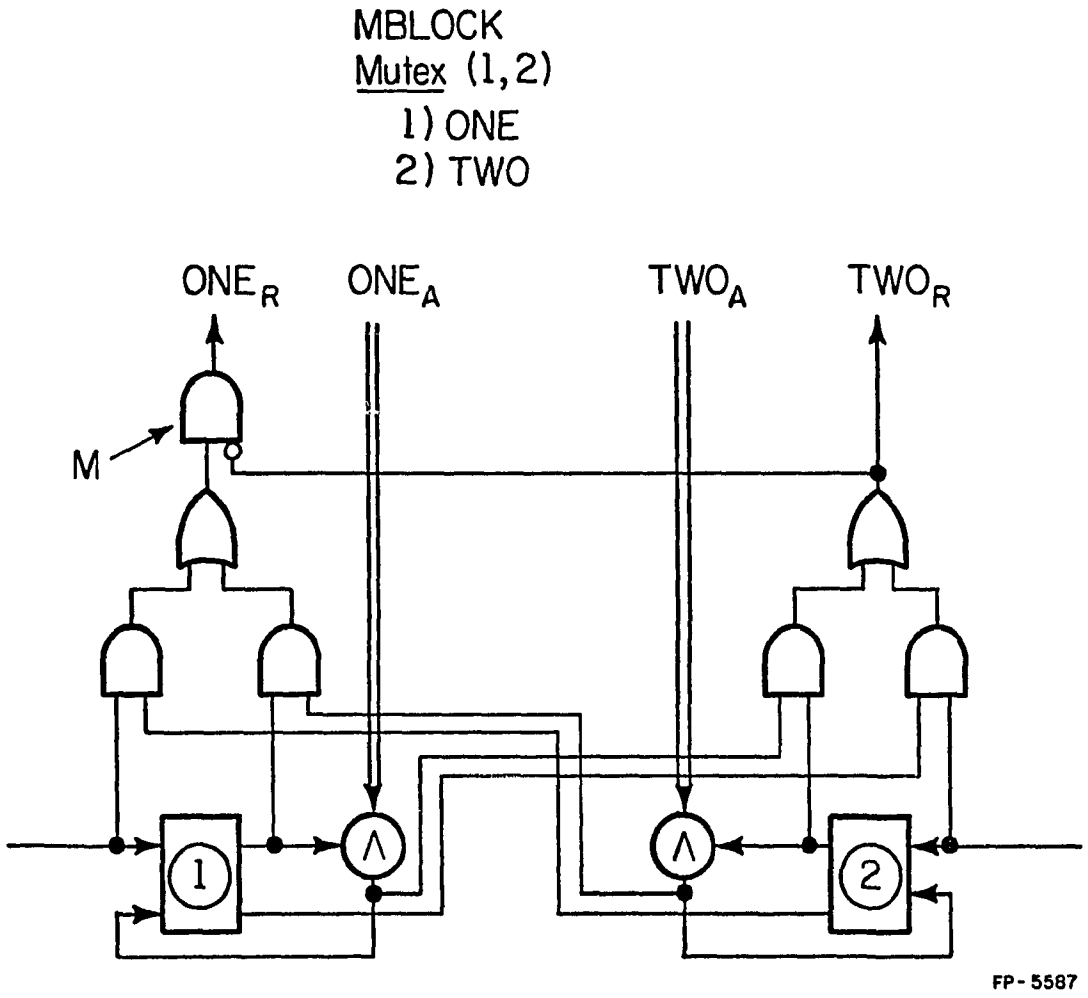


Figure 8.15. PA Implementation of a MPROC Block.

except that time is now defined discretely. The PN graph model of behavior still applies, as do the conclusions of Chapters 6 and 7. Hence PA implementations are also deadlock-free.

Finally, we make a concluding observation. If our finite state machine were designed using a state table, the notion of deadlock would not arise. It would be easy enough to ensure that no trap states exist. However, this method is all but impossible for any but the simplest machine. Hence, other representations and methods are required to design the finite state machine. With these the notion of deadlock arises. In a sense then, deadlock can be viewed as a function of the methods used to design and represent the machine.

9. COMPARISONS TO OTHER CHDLS AND OTHER APPLICATIONS

In this chapter other applications of some of the ideas developed in this thesis, as a result of specifying the CHDL, are discussed. Also, our approach to CHDLs is compared to others.

9.1 Other Applications

Several people have suggested the use of fork, join and quit operations (or their equivalent) for use in high level programming languages to enable programmers to write programs in which the potential for multiprocessing can be explicitly communicated to the compiler (see [Con 63], [And 65], [Opl 65] and [Den 66]). The use of fork is analogous to the effect of a W module on the flow of control, and the use of join and quit is analogous to the effect of a J module on the flow of control.

These operations allow the programmer to specify a control flow which can deadlock. By adopting a programming discipline similar to the one we have used in the syntax of the CHDL, such situations can be avoided.

Many programmers consider that the use of fork, join and quit in high level programming languages obscures the underlying algorithm, that a program specifies, by representing the algorithm in a non-sequential fashion (see [Wir 66]). To accommodate this criticism and still retain the capability of multiprocessing, it is necessary to automatically detect segments of a program that can be executed concurrently, and have some mechanism at the assembly language level, or at the firmware level, for expressing concurrency. If this mechanism uses operations similar to fork, join and quit, we can again impose a discipline on usage to ensure that control flow does not deadlock. As a footnote to this discussion on programming language

constructs that facilitate multiprocessing, it is interesting to note that the MPROC block of the CHDL is analogous to a simple form of monitor (see [Hoa 74]).

Finally, there are two obvious candidates for any design methodology that includes something similar to the CHDL. These are the RTMs (Register-transfer modules) of the Digital Electronics Corporation (see [Bel 72]), and the Macromodules of Washington University (see [Cla 67]). Both of these are sets of asynchronous modules which contain elements of both CS and DS, that can be interconnected to form custom systems. The types of CS that they can produce are similar to those possible with the CS modules of Chapter 2. Hence, there is a need for an interconnection discipline, that could be imposed by a CHDL, to ensure that control flow does not deadlock. In the case of the RTMs, some researchers have suggested a design methodology that involves designing the target system as an interconnection of RTMs, then analyzing the resulting control flow using PNs (see [Hue 75]). Such an approach leads, in general, to complex analyses just to confirm that the control flow is free of potential deadlock. A further drawback also results, in that such analyses do not indicate how to correctly redesign a system which has been found to have a potential deadlock.

9.2 Comparisons to Other CHDLs

There are currently no CHDLs that are suitable for specifying multiprocessing systems. The major weakness of present CHDLs, in this respect, is the very limited nature of the CSs that they can describe. As a case in point, consider two of the most popular CHDLs, viz. ISP (see [Bel 71]) and AHPL (see [Hil 73]). Both have only very simple CS constructs. To use either of them to describe overlapping or mutually exclusive processes would be awkward, as all of the coordination would have to be done through

a system of flags declared in the DS. Furthermore, they can only describe simple series/parallel type concurrency. Nevertheless, they could easily be improved, from a multiprocessing point of view, by adding a few appropriate constructs: semaphores; queues in the control flow; a more flexible method of representing concurrency. There is an early example of a CHDL which comes closer to being suitable for specifying multiprocessing systems, and that is the Computer Compiler (see [Met 66]). This, however, can describe systems with potential deadlock in their CS.

10. CONCLUSION

To recapitulate, the two major purposes of this thesis were:

1. To develop a CHDL with sufficient scope to describe multiprocessing systems.
2. To specify the CHDL so that SC programs describe systems which have deadlock-free CSs.

A CHDL was developed in Chapters 2, 3, and 4, and it was shown in Chapters 6 and 7 that it does, in fact, achieve these purposes. To motivate the use of the CHDL it was used to design a small system in Chapter 5. Actual gate level implementations, both asynchronous and synchronous, were discussed in Chapter 8. Chapter 9 discussed some extensions of the thesis and commented on other work.

One general point of note is the hierarchical nature of the CHDL that was pointed out throughout this thesis. This follows as a consequence of the observation made in Chapter 6, viz. that the CHDL satisfies the Structure Theorem of [Mil 72] and, hence, the Top Down Corollary: programs can be written or read top down. For the user this means there is a convenient relationship between the CHDL text (static) and the intended operation of the system it describes (dynamic).

In the system model of Figure 1.1 we viewed a digital system as composed of a CS and DS. This thesis has been concerned mainly with the CS aspects of a CHDL. Further research could be carried out on the DS aspects of a CHDL, with special reference to the needs of multiprocessing. As was noted in Section 5.3, a formalism for data type definition is needed that is suitable for hardware data objects (busses, registers, subfields of registers, etc.). Of particular interest would be a method which, aided by the

formalism for data definition, would facilitate the design of systems with deterministic DSs. There are two approaches that may be taken. The first is preventative, i.e. specify the CHDL so that it cannot describe non-deterministic systems. This is the approach that we have adopted in regard to deadlock. The second is curative, i.e. the DS is examined, after the design process, for sources of non-determinism. In the opinion of the author, prevention is better than cure, but it should not be undertaken to the point of limiting the scope of a CHDL until it becomes useless.

REFERENCES

Alt 69 Altman, S. M., and A. W. Lo, "Systematic Design for Modular Realization of Control Modules," 1969 SJCC, AFIPS Conf. Proc., Vol. 34, pp. 587-595, 1969.

Alt 70 Altman, S. M., and P. J. Denning, "Decomposition of Control Networks," Rec. of the Proj. MAC Conf. on Concurrent Systems and Parallel Computation, pp. 81-92, 1970.

And 65 Anderson, J. P., "Program Structures for Parallel Processing," CACM, Vol. 8, No. 12, pp. 786-788, Dec. 65.

And 67 Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo, "The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling," IBM Jour. of R&D, Vol. 11, No. 1, pp. 8-24, Jan. 67.

Aze 75 Azema, P., M. Diaz, and J. E. Doucet, "Multilevel Description Using Petri Nets," Proc. 1975 Int. Symp. on Computer Hardware Description Languages and their Applications, pp. 188-190, Sept. 75.

Bar 75 Barbacci, M. R., "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems," IEEE TC, Vol. C-24, No. 2, pp. 137-150, Feb. 75.

Bel 71 Bell, C. G., and A. Newell, Computer Structures: Readings and Examples, McGraw-Hill, New York, 71.

Bel 72 Bell, C. G., J. L. Eggert, J. Grason, and P. Williams, "The Description and Use of Register-transfer Modules (RTM s)," IEEE TC, Vol. C-21, No. 5, pp. 495-500, May 72.

Bru 71 Bruno, J., and S. Altman, "A Theory of Asynchronous Control Networks," IEEE TC, Vol. C-20, No. 6, pp. 629-638, Jun. 71.

Cat 66 Catt, I., "Time Loss Through Gating of Asynchronous Logic Signal Pulses," IEEE TC, Vol. EC-15, No. 1, pp. 108-111, Feb. 66.

Cha 73 Chaney, T. J., and C. E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," IEEE TC, Vol. C-22, No. 4, pp. 421-422, Apr. 73.

Cla 67 Clark, M. A., "Macromodular Computer Systems," 1967 SJCC, AFIPS Conf. Proc., Vol. 30, pp. 335-336, 1967.

Com 71 Commoner, F., A. W. Holt, S. Even, and A. Pnueli, "Marked Directed Graphs," J. Comput. Syst. Sci., Vol. 5, pp. 511-523, 1971.

Con 63 Conway, M. E., "A Multiprocessor System Design," 1963 FJCC, AFIPS Conf. Proc., Vol. 24, pp. 139-146, 1963.

Den 66 Dennis, J. B., and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," CACM, Vol. 9, No. 3, pp. 143-155, Mar. 66.

Den 70 Dennis, J. B., "Modular, Asynchronous Control Structures for a High Performance Processor," Rec. of the Proj. MAC Conf. on Concurrent Systems and Parallel Computation, pp. 55-80, 1970.

Den 71 Dennis, J. B., and S. S. Patil, "Speed Independent Asynchronous Circuits," Proc. 4-th Hawaii Int. Conf. on Syst. Sci., pp. 55-58, 1971.

Fal 64 Falkoff, A. D., E. E. Iverson, and E. H. Sussenguth, "A Formal Description of System/360," IBM Syst. Jour., Vol. 3, No. 3, pp. 198-262, 1964.

Fig 73 Figueroa, M. A., Analysis of Languages for the Design of Digital Computers, CSL Report R-611, May 73.

Fra 75 Franta, W. R., and W. K. Giloi, "APL*DS: A Hardware Description Language for Design and Simulation," Proc. 1975 Int. Symp. on Computer Hardware Description Languages and their Applications, pp. 45-52, Sept. 75.

Fri 67 Friedman, T. D., "ALERT: A Program to Compile Logic Designs of New Computers," Digest 1st Annual IEEE Comp. Conf., pp. 128-130, Sept. 67.

Fri 69 Friedman, T. D., and S. C. Yang, "Methods Used in an Automated Logic Design Generator (ALERT)," IEEE TC, Vol. C-18, No. 7, pp. 593-613, Jul. 69.

Glu 65 Glushkov, V. M., "Automata Theory and Structural Design Problems of Digital Machines," Kibernetica, Vol. 1, No.1, pp. 3-11, 1965.

Gsc 75 Gschwind, H. W., and E. J., McCluskey, Design of Digital Computers, Springer-Verlag, New York, 1975.

Hei 76 Heimerdinger, W. L., and L. A. Jack, "A Graph Theoretic Approach to Fault Tolerant Computing," 1975-76 Annual Report AFOSR Contract No. F44620-75-C-0053, Mar. 22, 1976.

Hil 73 Hill, F. J., and G. R. Peterson, Digital Systems: Hardware Organization and Design, New York, 1973.

Hoa 74 Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM, Vol. 17, No. 10, Oct. 74.

Hol 68 Holt, A. W., Final Report of the Information System Theory Project, Tech. Report RADC-TR-68-305, Rome Air Development Center, New York, 1968.

Hue 75 Huen, W. H., and D. P. Siewiorek, "Intermodule Protocol for Register Transfer Level Modules: Representation and Analytic Tools," Proc. 2-nd Annual Symp. on Computer Architecture, Houston, TX., Feb. 75.

Jum 73 Jump, J. R., and P. S. Thiagarajan, "On the Equivalence of Asynchronous Control Structures," SIAM Jour. Comp., Vol. 2, No. 2, pp. 67-87, Jun. 73.

Jum 74 Jump, J. R., "Asynchronous Control Arrays," IEEE TC, Vol. C-23, No. 10, pp. 1020-1029, Oct. 74.

Kel 74 Keller, R. M., "Towards a Theory of Universal Speed-Independent Modules," IEEE TC, Vol. C-23, No. 1, pp. 21-33, Jan. 74.

Knu 69 Knuth, D. E., The Art of Computer Programming, Vol. 1: Fundamental Algorithms, Addison-Wesley, Reading, MA, 1969.

Knu 73 Knuth, D. E., The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.

Knu 74 Knuth, D. E., "Structured Programming with goto Statements," Computing Surveys, Vol. 6, No. 4, Dec. 74.

Met 66 Metze, G., and S. Seshu, "A Proposal for a Computer Compiler," 1966 SJCC, AFIPS Conf. Proc., Vol. 21, pp. 121-129, 1966.

Mil 65 Miller, R. E., Switching Theory, Vol. II, John Wiley, New York, 1965.

Mil 72 Mills, H., Mathematical Foundation for Structured Programming, FSC72-6012, Federal Systems Division, IBM Corp., Gaithersburg, MD, Feb. 72.

Mud 75 Mudge, T., "Specifying a Design Language for Digital Systems," Proc. 13-th Annual Allerton Conf. on Circuit and System Theory, pp. 905-915, Oct. 75.

Mud 77 Mudge, T., A Design Language for Modular Asynchronous Control Structures, CSL Report R-759, Feb. 77.

Mul 63 Muller, D. E., "Asynchronous Logics and Application to Information Processing," Switching Theory in Space Technology, Stanford Univ. Press, Stanford, CA, 1963.

Opl 65 Opler, A., "Procedure-Oriented Language Statements to Facilitate Parallel Processing," CACM, Vol. 8, No. 5, May 65.

Pat 72 Patil, S. S., and J. B. Dennis, "The Description and Realization of Digital Systems," COMPCON 72, Proc. IEEE Comp. Conf., pp. 313-316, Sept. 72.

Pat 75 Patil, S. S., An Asynchronous Logic Array, Comp. Structures Group Memo 111-1, Proj. MAC, MIT, Feb. 75.

Pet 66 Petri, C. A., Communication with Automata, Suppl. 1 to Tech. Report RADC-TR-65-377, Vol. 1, Rome Air Development Center, New York, 1966.

Pet 73 Peterson, J. L., Modelling of Parallel Systems, Ph.D. Thesis, Dept. E.E. Stanford Univ., Stanford, CA, Dec. 73.

Pet 74 Peterson, J. B., On High Level Digital System Design, CSL Report R-653, Jul. 74.

Pro 75 Proc. 1975 Int. Symp. on Computer Hardware Description Languages and their Applications, IEEE, New York, 1975.

Sel 68 Sellers, F. F., M. Y. Hsiao, and L. W. Bearnson, Error Detecting Logic for Digital Computers, McGraw-Hill, New York, 1968.

Smi 77 Smith, F. M., Creating Simulators from a Design Language, CSL Report R-773, Jul. 77.

Tom 67 Tomasulo, R. M., "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," IBM Jour. of R&D, Vol. 11, No. 1, pp. 25-33, Jan. 67.

Ung 69 Unger, S. H., Asynchronous Sequential Switching Circuits, John Wiley, New York, 1970.

Wir 66 Wirth, N., "A Note on 'Program Structures for Parallel Processing'," CACM, Vol. 9, No. 5, May 66.

VITA

Trevor Nigel Mudge was born in London, England, on November 28, 1947. He received a B.Sc. degree in Cybernetics from the University of Reading in England in 1969, and an M.S. degree in Computer Science in 1973 from the University of Illinois. He was a research assistant with the Information Engineering Laboratory at the Department of Computer Science from 1969-1974. From 1974-1977 he worked with the Digital Systems Group at the Coordinated Science Laboratory.